Ingegneria del Software

Progetto

Insieme di *attività* e *compiti* con le seguenti proprietà:

- Raggiungano determinati obiettivi con specifiche fissate
- Hanno date d'inizio e fine ben definite
- Contano su disponibilità limitate di risorse (tempo, fondi, denaro, strumenti)
- Nel loro svolgimento consumano risorse.

(Definizione di H. Kerzner)
Attenzione: Attività e compiti non sono la stessa cosa

Attività e compiti

Le attività sono le cose che vanno fatte (in senso generale), mentre i compiti sono attività assegnate a qualcuno.
Tra le attività che devono essere svolte nell sviluppo di un progetto troviamo:

Pianificazione: Gestire risorse e responsabilità

Analisi dei requisiti: definire **cosa** si deve fare

Progettazione: (design) definire **come** farlo

Realizzazione: farlo:

- Perseguendo **qualità**
- **Verificando** l'assenza di errori
- **Validando** il risultato rispetto alle attese

Ingegneria

"Applicazione di principi scientifici e matematici a fini pratici" (American Heritage Dictionary)
L'ingegneria *applica* principi noti e autorevoli, non ne inventa $\Longrightarrow$ Best practice
Il fine pratico è spesso di carattere civile e sociale, comportando responsabilità di carattere etico e professionale.

Prodotto Software

Si può definire prodotto software uno dei seguenti:

Commessa Forma, contenuto e funzione sono fissate dal committente

Pacchetto Forma, contenuto e funzione sono adatte alla replicazione

Componente Forma, contenuto e funzione sono adatte alla composizione

Servizio Forma, contenuto e funzione sono fissate dal problema

Ciclo di vita

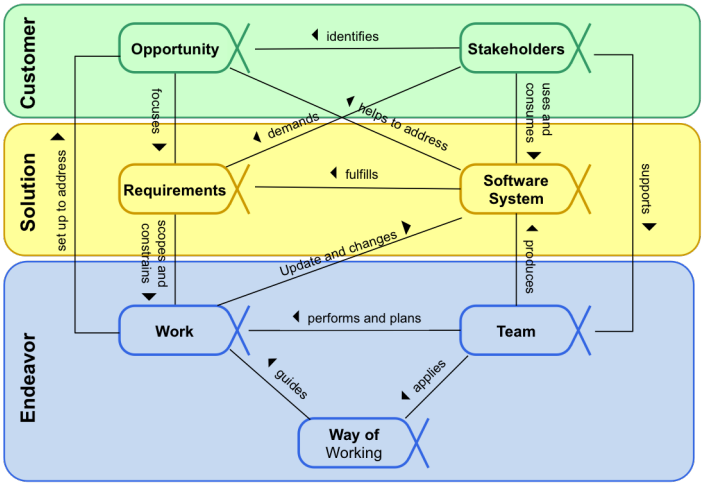
Gli stati di avanzamento che il prodotto assume dal concepimento al proprio ritiro

Efficacia

Misura la capacità di raggiungere gli obiettivi fissati $\Longrightarrow$ grado di conformità

Efficienza	**Misura** l'abilità di raggiungere gli obiettivi impiegando meno risorse possibili $\Longrightarrow$ riduzione dello spreco
Best Practice	Modo di fare noto che abbia dimostrato di garantire i migliori risultati (in termini di efficienza ed efficacia) in circostanze specifiche e note
Utilità di un software	Un software si dice tanto più utile quanto più è usato
Metrica	Integrale degli usi (o utenti) nel tempo, misura dell'utilità di un software
Manutenzione	Nei software con ciclo di vita "lungo", viene effettuata manutenzione (con i propri costi annessi): **Correttiva** Atta alla rimozione di difetti **Adattativa** Raffinamento dei requisiti **Evolutiva** Evoluzione del sistema
Cos'è L'ingegneria del Software	Nata nel 1968, ha l'obiettivo di Raccogliere, organizzare e consolidare la conoscenza necessaria a realizzare progetti software con massima efficienza ed efficacia. La materia non ha una base teorica certa, dato che è una disciplina molto giovane.
Secondo il Glossario IEEE	L'approccio • **Sistematico** – Modo di lavorare metodico e rigoroso – che studia, fa uso ed evolve le best practice di dominio • **Disciplinato** (Che segue regole fissate) • **Quantificabile** (Che permette di misurare efficienza ed efficacia) allo sviluppo, l'uso, la manutenzione ed il ritiro del software.
Metodo di studio	Si va a costruire incrementalmente il proprio glossario 1. Basandolo inizialmente sulla teoria (individuando termini e significati) 2. Consolidandolo con la prativa (applicazione e confronto critico con l'esperienza) 3. Discutendo tale glossario con i colleghi (unione di conoscenze parziali, correzione reciproca). Inoltre è necessario integrare le diapositive date nel corso con fonti e risorse autorevoli, in sessioni di studio personale.

Elementi di un progetto (gli "alphas" secondo SEMAT)



Opportunità

Set di circostanze che rendono appropriata la produzione o il cambiamento di un sistema software.
In pratica è la ragione per la creazione di un nuovo (o diverso) sistema software. Rappresenta la comprensione condivisa delle necessità dello stakeholder (portatore di interesse) ed aiuta a formare i requisiti per il nuovo sistema software, dando giustificazione alla sua creazione.

Ciclo di Vita

Stati che il prodotto assume dal concepimento al ritiro.
Il ciclo di vita del software è equiparabile ad una macchina a stati finiti.

Processi di Ciclo di vita

Way of working per abilitare le transizioni (corrette) tra stati del ciclo di vita (**come** fare la transizione)

Modelli di ciclo di vita

Astrazione che rappresenta i possibili cicli di vita.
Un professionista deve sapere **prima** la successione di attività del ciclo di vita in modo da stimare oneri e tempi associati al progetto software.
Comprendono tutto il ciclo di vita (comprese le fasi d'uso e manutenzione), noi ci occuperemo solo del "ciclo di sviluppo".
Alcuni esempi di modelli:

- Sequenziale
- Incrementale
- A spirale
- Agile

Prototipo

"Versione Iniziale", o nel nostro caso una versione che:

- Serve per **capire meglio**
- Può essere "usa e getta" (capisco e poi scarto)
- Può essere un pezzo del software che resta in via permanente
- Possono rappresentare stati di avanzamento (vedi "incremento") certi nello sviluppo del software (baseline)

Incremento

Procedere per aggiunte su una base

Iterazione

Procedere per rivisitazioni (può includere un incremento o addirittura un decremento).
L'iterazione è un processo di durata non determinabile (anche infinita)

Riuso

In un progetto software moderno è importante fare riuso di algoritmi, tecniche, librerie e codice già esistente. Può essere:

Opportunistico copia/incolla senza comprensione; non necessariamente ripetibile

Sistematico ricerca, comprensione e riuso del codice. Ripetibile.

Manutenzione

Un buon software deve essere mantenuto, non necessariamente per riparare problemi ma anche a scopo evolutivo.
Le cose che funzionano oggi potrebbero non funzionare domani, quindi è necessario mantenere la storia dell'evoluzione/manutenzione/sviluppo del software, che viene fatto tramite il controllo di versione (versioning)

Componenti

Un software non è monolitico, ma diviso in parti che sono messe insieme secondo una logica descritta nella **configurazione**
La configurazione è stabilita tramite strumenti di controllo di configurazione.
L'azione che automatizza la procedura descritta nella configurazione è detta build

Processo

Insieme di attività **correlate** e **coese** che trasformano ingressi (bisogni) in uscite (prodotti) secondo regole date, consumando risorse nel farlo.
Non è la stessa definizione di progetto, infatti è un pezzo del progetto.
Le regole date sono atte a garantire efficienza ed efficacia.
Efficenza ed efficacia vanno misurate in corso d'opera ottenendo **dati** misurabili dell'attività e conseguentemente si impongono vincoli per mantenere efficienza ed efficacia.

Misurazioni

Le misurazioni devono essere:

- Tempestive
- Accurate
- Non intrusive (non devono infastidire la gente)

Coesione

Si dice coeso un insieme di parti che concorrono ad un unico obiettivo (esempio le classi dei programmi)

Correlazione

Si dicono correlati degli oggetti che hanno un motivo per stare assieme.
Fa da premessa alla modularità.

Metriche

Metodi di misurazione.
Metrica di efficienza: Produttività; definita come

$$\frac{q.tà\ di\ prodotto\ realizzato}{risorse}$$

Metrica di efficacia (o di conformità): Il grado di raggiungimento degli obiettivi

- Interni (del fornitore)
- Esterni (gradimento dell'utenza)

Economicità

Principio base del lavoro
Riassumibile come l'insieme di efficienza ed efficacia.

Nascita dei processi

I processi sono nati dai committenti insoddisfatti del lavoro dei fornitori.
Sono nate iniziative per vincolare i fornitori a certe regole.
Noi ci occuperemo di uno standard generale (esistono anche standard settoriali).
Prime applicazioni dei processi: USA (dipartimento della difesa)

Visioni

Uno standard può essere visto in due modi

- Riferimento a cui vorremmo aderire (best practice)
- Riferimento a cui dobbiamo assolutamente aderire (Standard imposto) $\Longrightarrow$ Modello di azione (opposto a "modello di valutazione")

A volte uno standard può essere sia un modello d'azione che un modello di valutazione.

Standard ISO/IEC 12207:1995

Generato dall'unione di più standard (soprattutto del Dipartimento della Difesa).
È uno standard **astratto**, parla dei processi da usare laddove esiste un ciclo di vita.
Esistono 3 macroclassi di processi

Processi Primari Esistono necessariamente laddove esiste un progetto

Processi Organizzativi A supporto del lavoro collettivo. Anche attraverso più progetti. Devono esistere affinchè l'organizzazione sia produttiva

Processi di Supporto Aiutano i processi primari ed organizzativi

Processi Primari

- Acquisizione
- Fornitura
- Sviluppo
- Uso/Operazione
- Manutenzione

Noi ci occuperemo dei primi 3.

Processi di Supporto

- Documentazione
- Gestione della configurazione
- Verifica (Tramite misurazioni)
- Validazione
- ...

Processi Organizzativi

- Gestione (Qui finiscono e si fanno le misurazioni così da mantenere affidabilità e non invasività)
- Training/Formazione
- ...

Modifica Dello Standard

Nel 2008 lo standard 12207 è stato esteso dato che il software è divenuto sempre più diffuso, fino ad arrivare a controllare sistemi al di fuori dei computer. Quindi si è aggiunto parte dello standard di qualità riguardante i sistemi.

Sistema

Aggregato di cose (hardware, software e persone) che hanno un'unico fine comune

Flusso Decisionale

Governato dall'organizzazione, istanziato in un progetto che a sua volta fa uso dell'engineering.

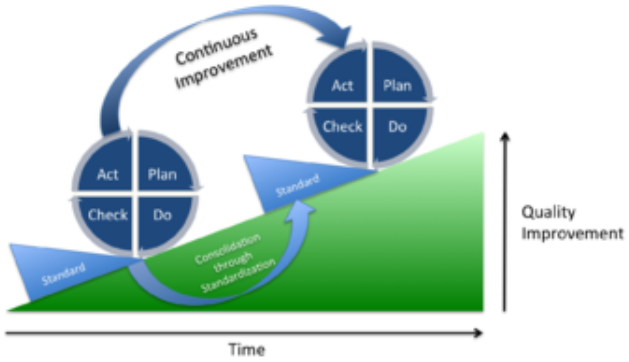
Organizzazione

Vi sono solitamente tanti settori dell'azienda, ognuno col suo modo di lavorare.
Il compito dell'organizzazione è prendere uno standard generico ed adattarlo alla propria azienda.
Un progetto, a sua volta, prende i processi aziendali e li istanzia in sè, nella forma di processi di progetto.
Questo processo è detto specializzazione di processi.
L'organizzazione interna dei processi dovrebbe essere incentrata sul principi di miglioramente continuo.

Miglioramento Continuo

- Pianifico
- Eseguo secondo i piani
- Valuto l'esito delle azioni di miglioramento
- Agisco standardizzando ciò che è andato bene e correggendo le carenze.

Questo viene fatto ciclicamente.



Lo standard funge da "cuneo", impedendo di peggiorare, mentre il miglioramento continua a spingere più in là la qualità. La standardizzazione "sposta" il cuneo verso un nuovo punto massimo.

Ciclo di vita

Solitamente visto come una macchina a stati, in cui si vede la successione di 4 stati:
$Concezione \Longrightarrow Sviluppo \Longrightarrow Uso \Longrightarrow Ritiro$
Ogni progetto si colloca su un tempo di calendario finito, lineare e segmentabile.
Ogni segmento temporale è concettualmente detto **fase**.
Il modo in cui mi muovo tra le fasi è deciso dal modello di ciclo di vita.

Code'n'Fix

È un **non-modello**, dove prima creo del codice, vedo se funziona ed eventualmente correggo i difetti.
Non ha organizzazione preordinata e porta a progetti caotici e difficilmente gestibili.

Modello Sequenziale

(W. Royce, 1970)
Detto anche "modello a cascata".
È un modello incentrato sulla ripetibilità del modello stesso e delle proprie fasi (in progetti diversi)
È un modello composto da fasi **rigidamente sequenziali**:

- Nessun Parallelismo
- Nessuna Sovrapposizione
- Nessun Ritorno a fasi precedenti

È un modello guidato da documentazione, in cui il software arriva solo alla fine. Fa fortissimo uso di pre- e post-condizioni.

Modello Sequenziale di Ciclo di Vita (Secondo lo standard ISO 12207)

Nello standard, le fasi del ciclo di vista del software sono viste come:
$Analisi \Longrightarrow Progettazione \Longrightarrow Realizzazione \Longrightarrow Manutenzione$
Dove il software arriva solo nella fase di Realizzazione.
Le prime 3 fasi sono dette "Modello sequenziale di Sviluppo"

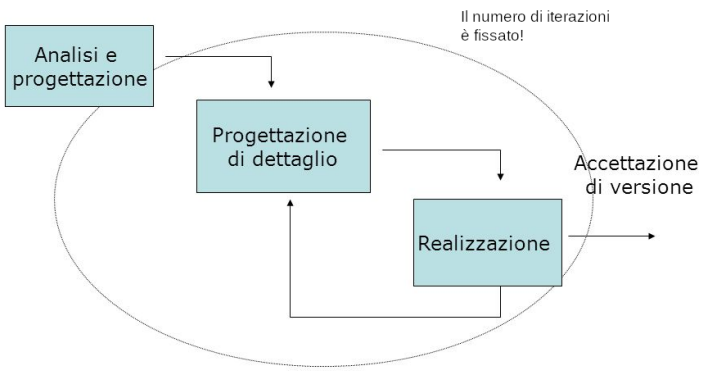
Caratteristiche del Modello Sequenziale

- È un modello iperdisciplinato
- La quantificabilità (delle misure temporali, di costo, ...) dipende fortemente dall'esperienza
- Sistematico
- Troppo Rigido

Modello Incrementale

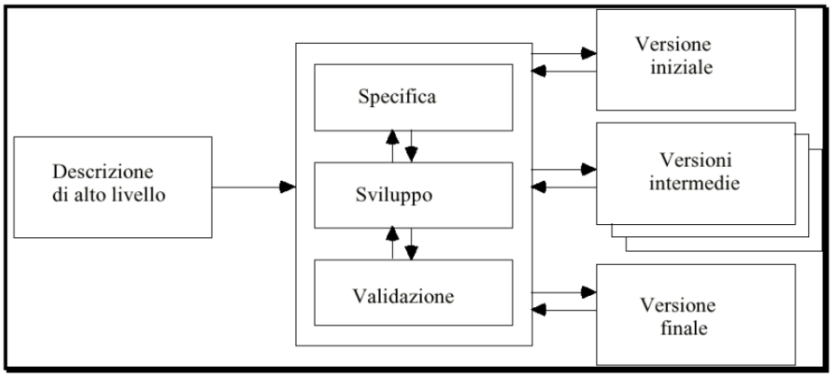
Spesso **non** conviene integrare tuttle le parti nelle fasi iniziali (cosiddetta "big bang integration"), dato che questo renderebbe arduo il troubleshooting in caso qualcosa non andasse. $\Longrightarrow$ conviene affidarsi ad una integrazione incrementale.
Il modello Incrementale prevede molteplici rilasci successivi.
Esso inizia analizzando il problema in maniera macroscopica; successivamente si decide quali incrementi fare per soddisfare gli obiettivi più importanti.
In questo modo le caratteristiche principali sono pronte prima e quelle minori hanno più tempo per stabilizzarsi ed armonizzarsi con il resto.
Successivamente decido come disegnare il modo in cui le componenti andranno ad integrarsi; dopodichè si inizia a sviluppare gli incrementi.
Se è possibile è conveniente parallelizzare degli incrementi (per esempio sviluppo di interfaccia e backend di un servizio online), invece di avere fasi strettamente sequenziali.
Le fasi di Analisi e Progettazione non si ripetono

Modello Incrementale secondo ISO 12207



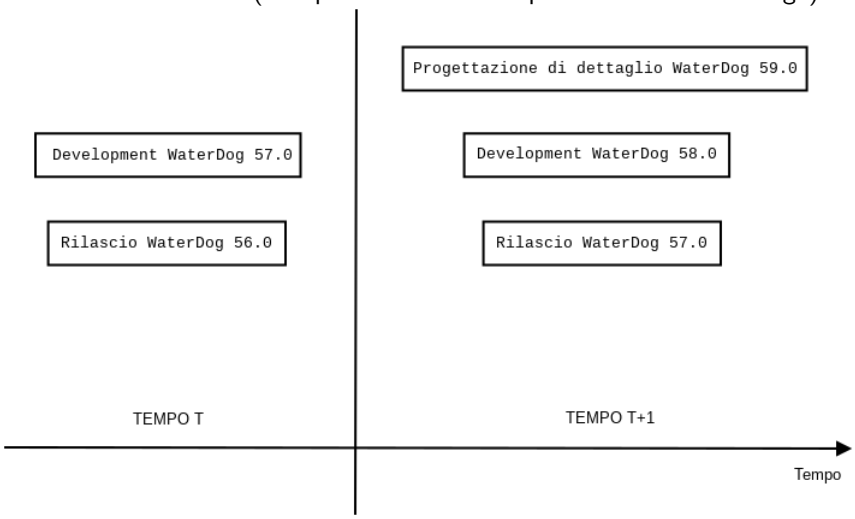
Modello evolutivo

Col tempo, i bisogni tendono a flettere, crescere o comunque cambiare con l'uso, quindi non è sempre possibile conoscere l'outline del problema specificatamente
Il modello evolutivo è composto da cicli paralleli su più versioni (stable, beta, alpha, . . .)



Caratteristiche del Modello Evolutivo

Il modello evolutivo presenta una fortissima sovrapposizione di attività diverse fatte su versioni diverse, un esempio potrebbe essere il ciclo di rilasci di un browser (che qui ho chiamato simpaticamente "WaterDog")



Modello a componenti

Basato sul riuso di componenti proprie e di terzi, arrivando persino ad adattare i requisiti in modo da renderli più adatti ad una progettazione a componenti con riuso.
È un modello che viene sempre più utilizzato data l'abbondanza di componenti (librerie, per esempio) software.

Modelli Agili: La nascita

I modelli agili nascono come reazione all'eccessiva rigitidà data dai modelli prevalenti fino alla fine degli anni '90

Modelli Agili: Il Manifesto

Riferimento: http://agilemanifesto.org/
Essenzialmente i modelli agili si basano su questi principi:

- Le regole troppo rigide non fanno bene
- La documentazione non è tanto importante quanto un software funzionante
- Invece di negoziare con lo stakeholder, collaboraci
- Essere pronti a reagire a situazioni in arrivo piuttosto che avere un piano

Modelli Agili: Critiche al manifesto

È necessario rendersi però conto che alcuni punti, così come sono, sono problematici:

- Un software senza documentazione è un costo più che un valore: non basta commentare il codice, è necessario spiegare e motivare certe scelte realizzative.
- Senza un piano non è possibile valutare i rischi e gli avanzamenti del processo
- È bene essere in grado di reagire, ma è soprattutto necessario essere consapevoli dei costi e benefici che certi cambiamenti comportano

User Story

I modelli Agili si basano sulle "User Story", dei documenti di descrizione del problema in esame, creati dialogando con l'utente, ascoltandolo, ragionando e proponendo soluzioni e miglioramenti in maniera attiva.
Inoltre definisce la strategia per confermare la conformità del software alle richieste del committente.

Esempi di Metodologie Agili

Scrum (Termine del football americano, un'azione apparentemente caotica che in realtà nasconde regole ed organizzazione)

Kanban (Giapponese, simile al just-in-time)

Scrumban

Scrum

Metodologia basata su iterazioni, con le seguenti componenti salienti:

Product Backlog Essenzialmente una todo list che contiene requisiti e funzionalità del prodotto

Sprint Si scelgono ed eseguonole fasi ritenute più utili al creare un "incremento utile", dura dalle 2 alle 4 settimane e si ottiene un prodotto potenzialmente vendibile

Sprint Backlog Essenzialmente una todo list che raccoglie l'insiemen di storie per il prossimo sprint

Fasi dello Scrum	**Sprint Planning** Pianificazione dello sprint **Daily Scrum** Un controllo giornaliero dell'avanzamento, nella forma di un incontro stand-up di circa 15 minuti. $\Longrightarrow$ anche se temporalmente breve è invasivo **Sprint Review** Si controllano i prodotti dello sprint **Sprint Retrospective** Si effettua un controllo qualità sullo sprint
Ciclo di Vita secondo SEMAT	Se all'interno di ogni componente di progetto SEMAT (opportunity, ...) si identificano delle sottofasi, si arriva ad una way of working ripetibile ed adattabile. Da un forte senso di "come facciamo ad avanzare", grazie alle tabelle di progressione.
Gestione Progetto	Insieme di attività (precisamente è un processo: project management) che governa tutto ciò che facciamo nel progetto. Appartiene alla categoria dei processi organizzativi.
Tempo di Calendario VS Tempo come risorsa	Questi due "tempi" sono molto diversi, mentre il tempo di calendario è semplice da calcolare, il tempo come risorsa dipende fortemente dalla pianificazione (un esempio è il tempo-persona)
Funzione Aziendale	Una funzione permanente, indipendente dal progetto in corso di svolgimento
Ruolo Di Progetto	Funzione aziendale assegnata a progetto
Ruoli principali	**Sviluppo** Responsabilità tecnica e realizzativa **Direzione** Responsabilità decisionale **Amministrazione** Responsabilità nella gestione di tecnologie di supporto ai processi **Qualità**
Ruoli a Progetto: Analisti	Si occupano di capire il problema. Deve fare molte attività per chiarire lo scopo del progetto, quindi ha molta influenza sul successo di quest'ultimo. Generalmente sono pochi, e raramente seguono il progetto fino alla fine
Ruoli a Progetto: Progettisti	Si occupano dell'aspetto tecnico della soluzione dei problemi spiegati dagli analisti. Definisce la miglior soluzione in termini di efficienza ed efficacia. Solitamente sono pochi ed accompagnano il progetto fino alla fine.

Ruoli a Progetto: Programmatori

Eseguono ciò che è stato definito dai progettisti e nulla di più.
Hanno competenze, visione e responsabilità molto circoscritte.
Sono molti.

Ruoli a Progetto: Verificatori

Si occupano della verifica di conformità del progetto.
Durano per l'intera durata di quest'ultimo.

Ruoli a progetto: Responsabile

Accentra le responsabilità di scelta ed approvazione.
Dura tanto quanto il progetto.
Ha responsabilità di:

- Pianificazione
- Gestione delle risorse umane
- Controllo, coordinamento e relazioni esterne
- Valutare richi, scelte ed alternative.

Ruoli a Progetto: Amministratore

Ruolo di supporto (conosciuto anche come SysAdmin)
Ha il controllo dell'ambiente di lavoro.

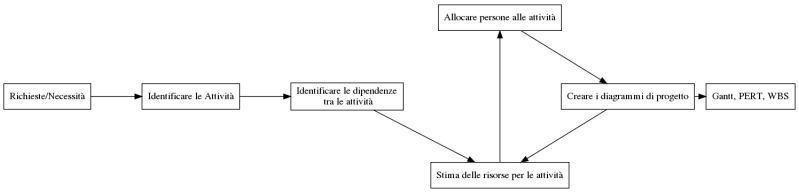
Ruoli a Progetto: Gestione Qualità

Ha una propria "way of working", quindi non cambia da progetto a progetto $\Longrightarrow$ Funzione aziendale.
Ha lo scopo di massimizzare efficienza ed efficacia.

Pianificazione

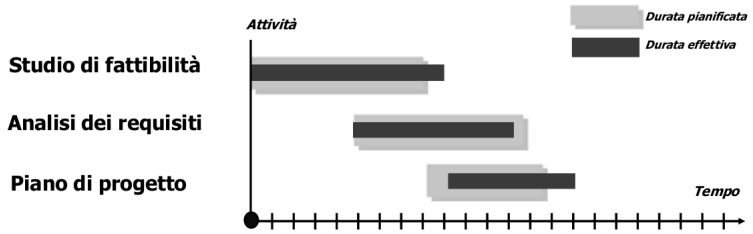
Organizzare il tempo di calendario assegnandoli ad attività e persone.
Viene fatta tramite appositi strumenti (e non su carta):

- Diagrammi di Gantt
- PERT/CPM
- WBS



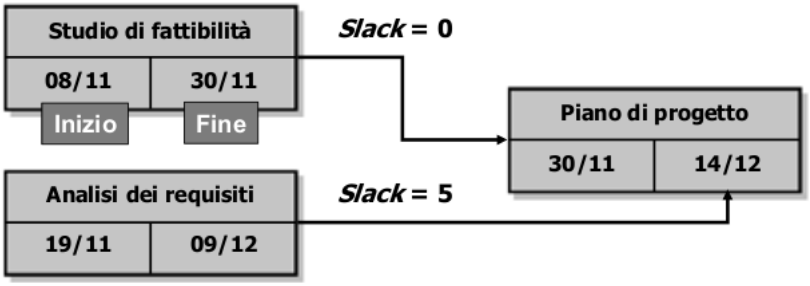
Diagrammi di GANTT

Il diagramma di Gantt (H.L. Gantt - 1917) è un diagramma di supporto alla gestione dei progetti, esso è costituito da un asse orizzontale, che rappresenta il tempo totale del progetto, suddiviso in fasi incrementali (giorni/settimane/mesi) e da un asse verticale che rappresenta le attività di progetto.
Vi sono poi delle barre orizzontali che rappresentano visivamente le sequenze, la durata e l'arco temporale di ogni attività del progetto, dando l'idea di quante attività in parallelo si stanno svolgendo ed eventualmente i punti in cui la parallelizzazione è migliorabile.
Man mano che il progetto prosegue, vi possono essere delle barre secondarie atte a rappresentare attività sottostanti.
Questo diagramma però non tiene conto dell'eventuale interdipendenza di attività che dovranno essere rappresentate in altro modo.
Inoltre è possibile sovrapporre le barre, in modo da avere visivamente sotto mano la differenza tra la durata pianificata e la durata effettiva del progetto.



PERT/CPM

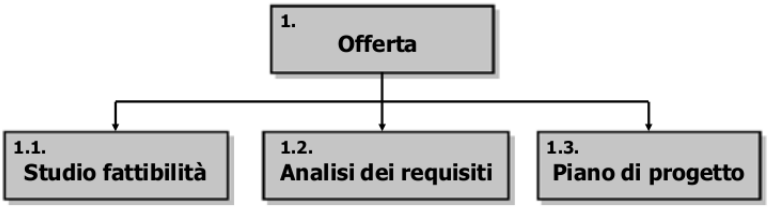
Il PERT (Program Evaluation and Review Technique), conosciuto anche come stima a tre valori, è un metodo statistico per determinare i tempi delle attività di progetto, prendendo conto dei valori di stima nel caso ottimale, probabile e peggiore; in modo da valutare in modo adeguato tempi e costi delle attività in caso di Incertezza.
CPM (Critical Path Method) è invece un metodo basato su grafi (detti diagrammi reticolari) per determinare la durata minima di un progetto, individuando le attività critiche che lo rappresentano.
Usando questi die metodi è possibile ragionare sulle scadenze di un progetto ed individuare la sequenza di attività con prodotto importante e dipendenze temporali stresse (Cammino critico)



Per slack si intende il margine temporale tra due attività, ovviamente uno slack negativo non è possibile altrimenti vorrebbe dire che vado ad assegnare personale e risorse ad un'attività senza che l'input per tale attività sia pronto.

WBS

Work Breakdown Structure
Ogni attività ha sottoattività non necessariamente sequenziale, ma comunque ben (univocamente) identificate



Problemi nell'allocazione delle risorse	Bisogna fare attenzione ad evitare queste trappole durante l'assegnazione di risorse: • Sottostimare le necessità • Sovrastimarle
CoCoMo	Constructive Cost Model Essenzialmente una formula per valutare le risorse necessarie, esprimendole in mesi-persona. Crea una serie di stime che possono essere rappresentate come curve.
Piano di Progetto	Documentazione letta da un verificatore, alcuni dati vanno condivisi con gli stakeholder e poi, una volta consolidato, viene visto dai membri del team. Contiene scopo, organizzazione del progetto e soprattutto analisi dei rischi
Rischi da Evitare	• Sforamento dei Costi • Sforamento dei Tempi • Risultati insoddisfacenti
Fattori Di Fallimento	• Requisiti Incompleti • Mancato coinvolgimento del cliente • Mancanza di Risorse • Fluttuazione dei requisiti
Fattori di Successo	• Coinvolgimento Del Cliente • Verità riguardo alla disponibilità del personale • Definizione chiara dei requisiti
Cos'è	Il Controllo Versione (o versioning) è un insieme di tecniche e strumenti atti alla gestione dei cambiamenti nei files. La necessità di poter gestire questi cambiamenti si è fatta sentire molto più nell'era dell'informatica, dove è abbastanza semplice introdurre modifiche ad un software che debbano poi essere rimosse, solitamente a seguito di malfunzionamenti provocati da tali modifiche.
Come Funziona il Versioning: Teoria	Sfruttando la teoria dei grafi possiamo vedere l'insieme di revisioni di un software come un grafo direzionato ed aciclico, in cui possiamo vedere le seguenti componenti: **Trunk** "Il tronco" Che rappresenta la linea di sviluppo principale, solitamente includendo nuove caratteristiche **Branches** "I rami" che si diramano dal trunk in diversi momenti della sua vita

Branches

I rami solitamente sono usati per

- Lavorare su una nuova feature senza intaccare la base su cui si basa il branch
- Lavorare parallelamente al trunk, un classico esempio è il branch di manutenzione di una certa versione del software
- Lavorare parallelamente ad altri branch, rendendo più rapido lo sviluppo del software

Merging

Quando un lavoro su un branch si considera concluso, vi sono due possibilità:

- Abbandono del branch (in tal caso si parla di "discontinued branch")
- Merge nel trunk

In caso di merge, il contenuto del branch viene unito a quello del trunk, con eventuale gestione di conflitti.

Commits

Essendo uno strumento anche collaborativo, i software di version control devono garantire l'atomicità delle operazioni, che in termini tecnici vengono dette "commit", in modo da evitare interferenze sui file che potrebbero rovinare il lavoro.

Software di Controllo Versione

Esistono vari software per fare controllo versione, i più famosi sono git, subversion (svn) e mercurial (hg).

Git

Nato nel 2005 ad opera di Linus Torvalds, git è un software di controllo versione nato dopo che molti sviluppatori del kernel linux hanno dovuto abbandonare l'accesso al codice sorgente tramite BitKeeper, dato che a detta del proprietario era stato fatto un reverse engineering dei protocolli Bitkeeper (cosa non ammessa)
Git è un sistema di versionamento con forte supporto allo sviluppo non lineare, permettendo branching e merging rapidi.
Inoltre è dedicato fortemente allo sviluppo distribuito e permette di pubblicare i repository via HTTP, FTP, ssh o rsync in maniera semplice.
Git è inoltre estremamente scalabile, è dotato di autenticazione crittografica della cronologia (così che se le vecchie versioni sono cambiate, questo possa essere notato), permette la concatenazione delle proprie componenti ed è dotao di strategie di fusione intercambiabili.

Subversion

SVN è un altro sistema di controllo versione, creato da CollabNet inc. come successore di CVS (Concurrent Versions System).
In questo programma le commit sono vere transazioni atomiche, infatti interrompere una commit lascia il repository in uno stato di incoerenza.
Alcune operazioni (come il branching) non dipendono dalla dimensione dei dati
È basato su un protocollo client/server che invia solo le differenze in entrambe le direzioni, rendendo i costi di comunicazione proporzionali alla dimensione di tali modifiche.

Conclusione Sul Versionamento

È assolutamente necessario porsi delle regole su chi deve intervenire su quale componente del progetto/documentazione, in modo da evitare perdite di tempo nella gestione dei conflitti.
La storia di un progetto è vitale, e deve essere conservata in maniera quanto più immutabile, se si devono operare modifiche in parti storiche del progetto, dovrebbe essere sempre possibile rintracciare la versione precedente a tale modifica.

Bug, Ticket Trackers e pianificazione

Un altro utile strumento di supporto per la coordinazione nei progetti è il tracker. Il cui scopo è quello, appunto di "tener traccia" di qualcosa.
Solitamente nell'ambito dello sviluppo software si usano bug trackers, per tener conto dei dettagli di bachi rilevati in un software ed i vari procedimenti, commenti e soluzioni proposte riguardo a tali bachi.
Un altro esempio, di carattere più generico sono i ticket trackers, dove si tiene conto non solo di bachi software ma anche di altre attività legate al progetto.
È comunque vitale avere una persona con l'esperienza necessaria, e la volontà di prendersi responsabilità delle decisioni prese sul progetto. Questo ruolo è solitamente coperto dal project manager.

UML

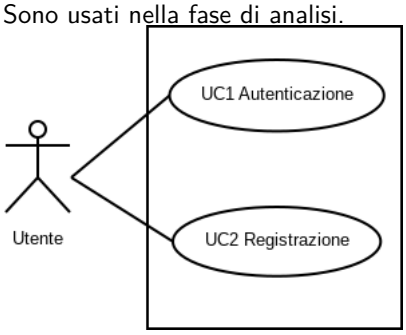
Unified Modeling Language
Insieme di notazioni grafiche (si usano disegni invece di sole parole in linguaggio naturale), pensato su un paradigma ad oggetti (object oriented).
Come Editor di riferimento in questo corso useremo Papyrus (http://www.eclipse.org/papyrus)

Uso di UML

- Come abbozzo (sketch): il modo più usato
- Come progetto: Inserendolo dentro la documentazione ed usandolo come base per il progetto
- Come linguaggio di programmazione: Così da poter generare codice automaticamente, partendo da UML.
  Così da avere un software provato (anche matematicamente)

Diagrammi dei casi d'uso

Sono usati nella fase di analisi.



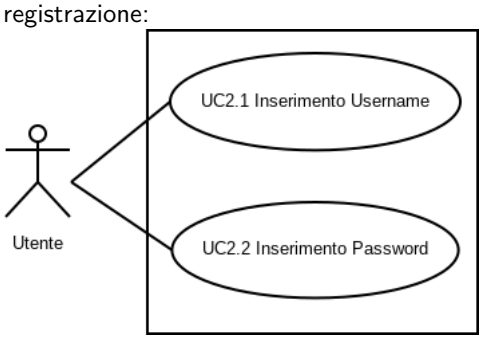
Legenda:

Attore o Ruolo: Chiunque agisca dall'esterno sul nostro software

Funzione/Caso D'uso Solitamente accompagnato da un identificatore (In questo caso UC1) a cui si fa riferimento nella descrizione.
Per relazionare attori e casi d'uso si usa una linea **continua non direzionata**

Dettaglio dei casi d'uso

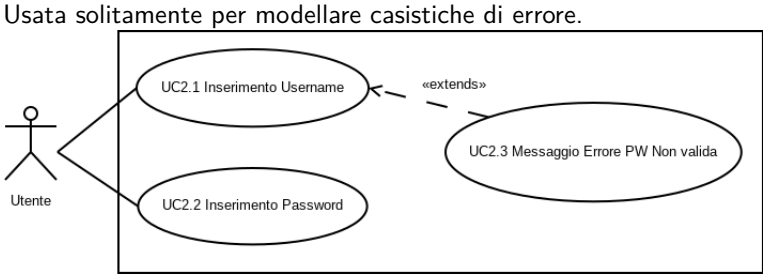
Posso poi entrare in maggior dettaglio, nei vari casi d'uso, come per la registrazione:



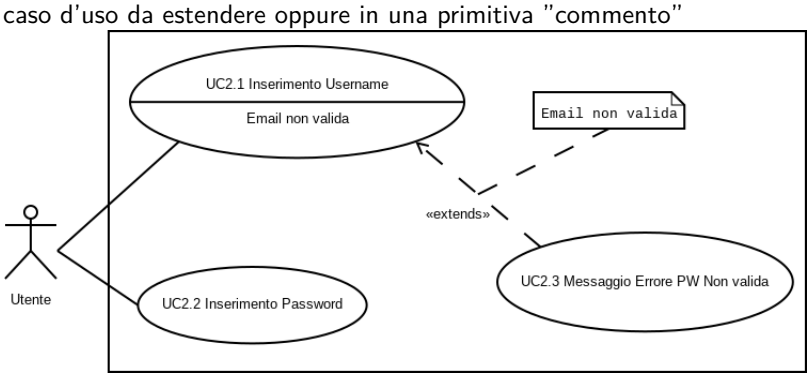
Notare come gli identificatori cambiano, in maniera gerarchica (UC2.1, che è evidentemente parte di UC2).
Quando sono arrivato al massimo livello di dettaglio, è semplice arrivare ai requisiti.

Estensione

Usata solitamente per modellare casistiche di errore.



Si relaziona con una **linea tratteggiata direzionata** verso la classe estesa.
La parola chiave ⟨⟨ extends ⟩⟩ è detta **direttiva UML**
L'esecuzione del caso d'uso da cui si estende **viene interrotta** e non arriva al termine.
Si aggiunge inoltre la condizione per arrivare all'estensione all'interno del caso d'uso da estendere oppure in una primitiva "commento"



Scenario

Sequenza di passi che descrivono interazioni.

Screnario Principale

Sequenza normale di operazioni che si eseguono "se tutto va bene"

Scenari Alternativi

Rappresentano scenari possibili diversi dallo scenario principale (ad esempio il messaggio di errore della mail non valida)

Casi D'uso

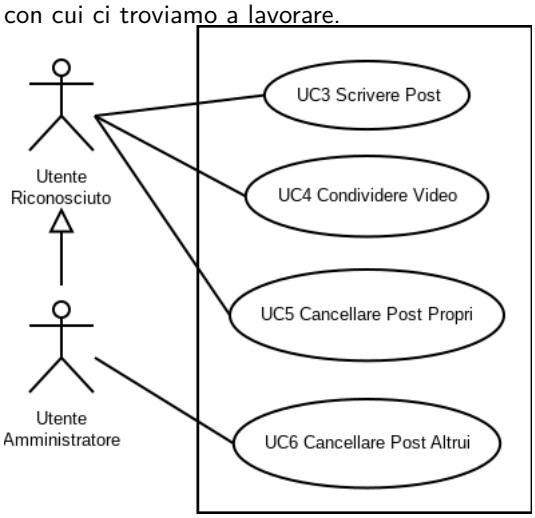
Specifiche testuali dei casi d'uso, molto più informative del diagramma. Specificano anche gli attori secondari, pre- e post-condizioni, descrizione dettagliata dello scenario principale e delle estensioni.
Esempio:

Caso d'uso: **UC1 - Registrazione**
Attore primario: **Utente**
Precondizioni: **L'utente non è ancora autenticato presso il sistema**
Postcondizioni: **L'utente possiede un'account presso il sistema, contraddistinto da una username e da una password**
Scenario principale:
1. L'utente accede al sistema
2. L'utente seleziona la funzionalità "Registrati"
3. L'utente inserisce una username univoca nel sistema
4. L'utente inserisce una password che rispetta i vincoli imposti

Estensioni:
a. Nel caso in cui l'utente inserisca una username già censita a sistema:
   1. L'utente non viene registrato presso il sistema
   2. Viene visualizzato un errore esplicativo
   3. Viene fornita all'utente la possibilità di scegliere un'altra password

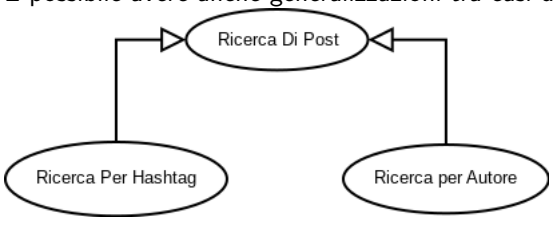
Attori

Persona o sistema esterno che interagisce con il nostro sistema.
In un sistema solitamente vi sono più attori, a seconda delle precondizioni con cui ci troviamo a lavorare.



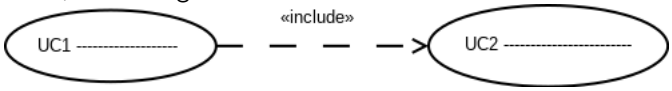
Subclassing (o Generalizzazione)

 con una freccia continua che termina in un triangolo vuoto, si definisce una generalizzazione tra attori (non è il subclassing della programmazione), nel caso visto prima possiamo vedere che l'utente amministratore ha le funzionalità di utente riconosciuto, più qualche altra.
È possibile avere anche generalizzazioni tra casi d'uso.



Inclusione

Ogniqualvolta un attore accede ad un caso d'uso, dopo che questo si conclude, viene eseguito **incondizionatamente** un altro caso d'uso.



La direzione della freccia tratteggiata va verso il caso d'uso eseguito.
Viene usato per seguire il principio DRY (Don't repeat yourself).

Analisi Dei Rischi

Devo trovare cosa può "rompere" la mia pianificazione. Vi sono 3 passaggi preliminari

1. Identificare i rischi nei quali potrei incorrere
   (Cosa potrebbe andare male, realisticamente. Include fattori personali, tecnologici, ...)
2. Per ogni rischio, analizzo quanto probabile e imminante questo rischio è, e quanto danno può provocare e quali conseguenze ci sono
3. Organizzo le attività allo scopo di minimizzare l'impatto e la probabilità dei rischi.

In più vi è un passaggio iterativo:

4. Controllo Continuativo
   Guardo ed aggiorno i rischi, raffinando la mia strategia

Pianificazione

Per ridurre i rischi, si pianifica su segmenti di tempo piccoli e controllabili, così da ridurre conseguenze, impatto e probabilità di questi.

Milestone

Punto **nel tempo** con significato strategico $\Longrightarrow$ punto di riferimento

Baseline

Punto di avanzamento **del prodotto** $\Longrightarrow$ È misurabile

Baseline+Milestone

La milestone va associata a risultati tangibili $\Longrightarrow$ ogni milestone è associata ad una o più baseline

Analisi dei Requisiti

Fase importantissima del progetto, oltre che prima attività del progetto software.

Requisito

- Un Bisogno (dal punto di vista del richiedente)
- Capacità di un sistema di adempiere ad una richiesta (visto dal punto di vista dell'offerente)

Verifica

Accertare che l'esecuzione di attività non introduca errori.
È un processo di supporto.
Si può ricordare con la frase "Did I build the system right?" (Ho fatto le cose come dovuto/nel modo giusto?)

Validazione

Accertare che il prodotto corrisponda alle attese.
Si può ricordare con la frase "Did I build the right system?" (Ho costruito il sistema giusto?)

Verifica E Baseline

Per sapere dove sono arrivato in una baseline faccio **molte verifiche** così da avere una misura di avanzamento

Qualifica

Composta dalle verifiche più la validazione finale (le validazioni sono molto costose, quindi non se ne fanno molte. Solitamente viene fatta solo la finale.)

Attività Necessaria in questa fase

- Analisi
- Piano di Qualifica

Inizio Dell'Analisi

- Studio dei bisogni e delle fonti
  Mi metto nell'ottica del committente per capire la natura e l'origine dei bisogni
  Identifico, specifico (univocamente) e classifico (per importanza, negoziabilità, ...) i requisiti.
- Modellazione concettuale del sistema
  Penso a come qualcuno da fuori agisce col mio sistema $\Longrightarrow$ Diagramma dei casi d'uso
  Vedo **cosa** fa il sistema per risolvere il problema, non **come** lo fa.

Processi di supporto Implicati

- Documentazione
- Gestione e manutenzione dei prodotti
  - È necessario gestire i requisiti (A scopo di fare controllo di conformità)
  - Gestione della configurazione
  - Gestione dei cambiamenti (È un processo, chiamato "Change management")

Prodotti Documentali

Non posso "raccontare a voce" informazioni collaborative perchè:

- "Verba Volant" - Non può rimanere traccia scritta di ciò che ho detto
- La discussione orale in stile meeting blocca le persone dal compiere le proprie attività

Quindi devo avere a disposizione dei formati non invasivi, cioè dei documenti scritti.

Primi prodotti documentali

Capitolato D'appalto Definisce i requisiti del cliente (Il "Cosa fare")

Specifica dei requisiti Software Il "Come fare" ciò che è definito dal capitolato
Comprende:

Studio di fattibilità Che è un documento riservato al fornitore

Analisi dei requisiti cioè definisco col cliente ciò che devo fare, con questo documento il fornitore viene formalmente ingaggiato per il lavoro

Ripartizione dei requisiti

Approccio Funzionale

È un approccio di tipo top-down.
Parte dal sistema completo (il tutto) e lo suddivide in componenti (secondo il principio "divide et impera")
È l'approccio più immediato per gli esseri umani.
Non aiuta il riuso, quindi è un pensiero da evitare.

Approccio Object-Oriented

Approccio di tipo bottom-up.
Parto da pezzi generici che opportunamente manipolati possono diventare il sistema.

Studio di Fattibilità

Valutazione dei costi, rischi e benefici.

- Fattibilità tecnico-organizzativa
- Rapporto costi/benefici
- Individuazione dei rischi
- Valutazione delle scadenze temporali
- Valutazione delle alternative:
  - Make Or Buy (Affidarsi al riuso oppure sviluppare ex-novo)
  - Scelte di Architettura
  - Strategie Operative

Questo passaggio ha tempistiche molto strette (Solitamente si ha un mese per Studio di Fattibilità **e** Analisi dei requisiti)

Tecniche di Analisi: Analisi dei bisogni e delle fonti

Ci si dedica alla comprensione del dominio
Mi metto nei panni dell'utente che ha scritto il capitolato, cercando di esplicitare ciò che è implicito nel dominio. (Possono infatti essere presenti requisiti impliciti)

Tecniche di Analisi: Interazione col Cliente

Interviste: quando capiamo cose col cliente, queste vanno trascritte nella cosiddetta "minuta col cliente", un documento scritto.
La minuta fa da appendice al capitolato ed ha valore contrattuale, quindi va approvato dalle parti.

Tecniche di Analisi: Discussioni Creative e collaborative

Brainstorming.
In ogni brainstorming ci devono essere $n + 2$ persone, di cui n discutono, più

- Qualcuno che funga da "scriba", raccogliendo i punti della discussione su cui il gruppo converge
- Qualcuno che regolamenti la discussione in modo che
  - Non ci si parli l'uno sopra l'altro
  - Non si abbia quella persona che "parla sempre"
  - Non si finisca per divagare

Tecniche di Analisi: Prototipazione

- Interna (Solo per il fornitore)
- Esterna (Per discussione col cliente)

Classificazione Dei Requisiti

Ordinare i requisiti facilita la comprensione, manutenzione e tracciamento.

- Attributi di prodotto
  "Cosa devo fare?"
  Riguardano strettamente il sistema/prodotto
- Attributi di Processo
  "Come devo farlo?"
  Riguarda il "way of working" (che viene solitamente imposto dal committente)

Altra possibile classificazione (per utilità strategica)

Obbligatori Irrinunciabili, per qualche stakeholder

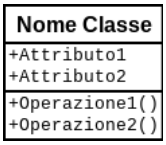
Desiderabili hanno valore aggiunto riconoscibile, ma non sono strettamente necesari

Opzionali relativamente utili, oppure contrattabili più avanti

Questi requisiti non devono essere in conflitto tra loro e devono essere **tracciabili** (avere requisiti atomici li rende facilmente verificabili, e quindi tracciabili)

Diagrammi delle classi

Descrivono il **tipo di oggetti** che fanno parte del sistema



L'unica parte obbligatoria è il nome della classe. Il resto sono dette "features".
Il blocco completo è detto, ovviamente, classe

Nota

Classe vs Oggetto

La classe è una "blueprint", che definisce come istanziare gli oggetti.
L'oggetto è **l'istanziazione** della classe

Attributi

Si definiscono in una sezione della classe, con la seguente sintassi:
`visibilità nome: tipo [molteplicità] = default {proprietà aggiuntive}`
Dove visibilità può essere

\+ Pubblica

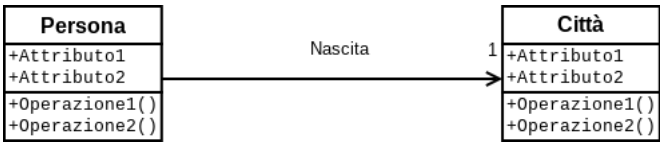
\- Privata

\# Protetta

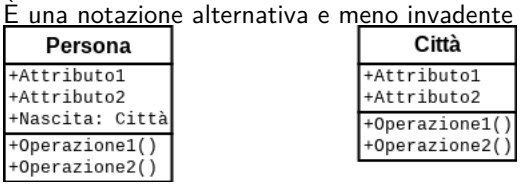
~ Di Package

Un esempio può essere: `#pippo: string`

Relazioni Fra Classi (Associazioni)



È una notazione alternativa e meno invadente di



solitamente quest'ultima è usata se B è un tipo primitivo, oppure se il diagramma è già molto confuso.
Anche se è possibile inserire la visibilità dell'associazione vicino al nome, questa viene solitamente definita private (in mancanza di simboli)

Molteplicità

- 1 (Sicuramente è presente una istanza)
- 0..1 (Nessuna o una sola istanza)
- * oppure 0..* (qualunque numero di istanze)
- 1..*
- [2,5] (da due a 5 istanze)

Proprietà aggiuntive

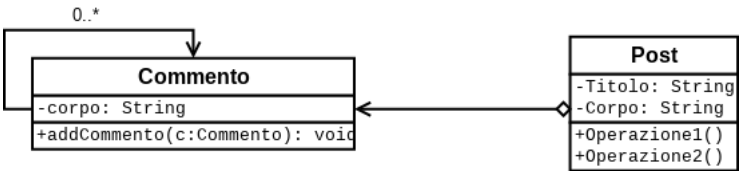
Solitamente le proprietà aggiuntive più usate sono

- Ordered (Per i vettori o gli array ad esempio)
- Unordered (Per i set)

Associazioni

Anche se le associazioni è etichettata con un verbo, bisogna usare nomi.
Evitare associazioni bidirezionali (che sarebbero delle dipendenze circolari.)

Esempio



Operazioni

Si scrivono sempre nel diagramma delle classi con la seguente sintassi:
`visibilità nome (lista parametri): ritorno proprietà`
Dove ogni membro della lista parametri ha la seguente sintassi:
`direzione nome:tipo=default`
Direzione può essere:

- in: Valore non modificato dal metodo (default)
- out: Valore che può essere modificato dal metodo, da evitare
- inout

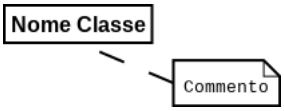
Un esempio può essere:
`+addCommento(in c:Commento): void`

Cavilli tecnici

Operazione Vs Metodo

Operazione e Metodo non sono la stessa cosa:
L'operazione è la "firma del metodo", quindi indipendente dall'implementazione singola.
Il metodo è l'implementazione/corpo dell'operazione.

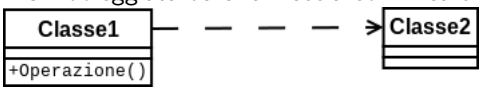
Commenti E Note



I commenti possono essere legati alla classe, ad un attributo o ad una operazione. Il loro scopo è aggiungere informazioni utili al diagramma.

Dipendenza

Si ha dipendenza tra due elementi se la modifica della definizione del primo può cambiare la definizione del secondo.
È simboleggiata da una freccia con linea tratteggiata



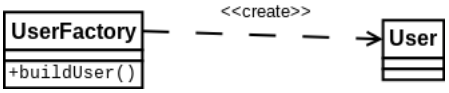
Le dipendenze vanno **minimizzate** (Loose Coupling).
Più codice condiviso c'è tra due tipi e maggiore è la dipendenza tra le classi.

Tipi di Dipendenze

- ⟨⟨ call ⟩⟩ - Invocazione di un'operazione nella classe di destinazione
- ⟨⟨ create ⟩⟩ - Creazione di istanze
- ⟨⟨ derive ⟩⟩ - Derivata di
- ⟨⟨ instantiate ⟩⟩ - È un'istanza di (vedi meta-classe)
- ⟨⟨ permit ⟩⟩ - Destinazione permette a source di accedere ai campi privati
- ⟨⟨ realize ⟩⟩ - Implementazione di una specifica interfaccia
- ⟨⟨ refine ⟩⟩ - Raffinamento tra livelli semantici
- ⟨⟨ substitute ⟩⟩ - Source è sostituibile a destinazione
- ⟨⟨ trace ⟩⟩ - Traccia i requisiti o come i combiamenti di una parte del modulo si collegano ad altre
- ⟨⟨ use ⟩⟩ - Source richiede Dest per la propria implementazione

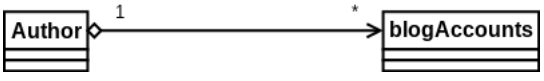
I più comuni sono i primi 2.

Esempio



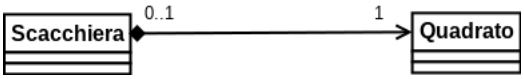
Aggregazione

Relazione "Parte Di"



Author è un "aggregato di blogAccounts", non può esistere un Author senza almeno un blogAccount.
Le aggregazioni possono essere condivise.

Composizione

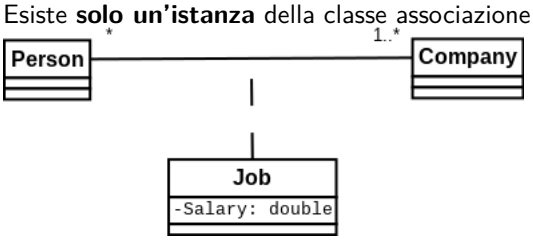


Come l'aggregazione, ma:

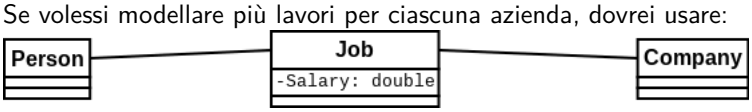
- Le parti appartengono ad **un solo aggregato**
- **Solo l'oggetto intero** (scacchiera) **può creare e distruggere** le sue parti (Quadrato)

Classi di Associazione

Aggiungono attributi e operazioni alle associazioni.
Esiste **solo un'istanza** della classe associazione fra le due classi.

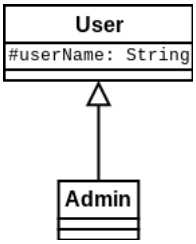


Una persona ha **un solo** lavoro in un'azienda
Questa dicitura mostra dei vincoli che non sarebbero visibili dal codice.
Se volessi modellare più lavori per ciascuna azienda, dovrei usare:



Generalizzazione

A generalizza B se ogni oggetto di B è anche un oggetto di A. Quindi abbiamo a che fare con una relazione IS-A.
Equivale all'ereditarietà dei linguaggi di programmazione.
L'ereditarietà multipla è possibile, ma **va evitata**



La classe derivata eredita dalla classe base, quindi i campi/operazioni ereditati non vanno inseriti nella classe derivata (a meno di override.)

Tracciamento dei Requisiti

Durante il tracciamento dei requisiti è importante che siano soddisfatte le condizioni di

Sufficienza Tutti i bisogni sono stati rilevati e trasformati in requisiti

Necessità Tutti i requisiti soddisfano un dato bisogno (Cioè ogni requisito ha una giustificazione della propria esistenza)

Documentazione dei requisiti

È fatta in un linguaggio misto. È importante

- Evitare le ambiguità interpretative (Usando norme che garantiscano una terminologia consistente)
- Usare linguaggi a diagrammi e formule (invece di testo e disegni "in stile libero")

Specifica Dei Requisiti: IEEE 830-1998

La specifica dei requisiti deve avere delle caratteristiche desiderabili:

- Non ambigua
- Corretta
- Completa
- Verificabile
- Consistente
- Modificabile
- Tracciabile
- Ordinata (Per rilevanza)

Verifica Dei Requisiti

Eseguita su un documento organizzato, viene effettuata tramite:

Walkthrough Traversata a pettine, lettura a largo spettro
Non so dov'è il problema, quindi vado a battere tuttle le strade per trovarlo

Ispezione Basata su una checklist mirata e solida (Si fa un'ispezione in stile "Gate Aeroportuale")

Gestione dei requisiti

È necessario che i requisiti siano contraddistinti da un **codice univoco ed informativo**
Solitamente sono ordinati in modo sequenziale sulla struttura del documento.
I requisiti devono essere ordinati in modo che inserimento, rimozione e gestione abbiano il minor impatto possibile.
Inoltre è vitare saper spezzare i requisiti monolitici e generali in requisiti più piccoli che siano immediatamente localizzabili, risolubili e verificabili.

Stati di Progresso Secondo SEMAT

Conceived Il comittente è stato identificato e gli stakeholder vedono sufficienti opportunità per il progetto

Bounded I macro bisogni sono chiari ed i meccanismi di gestione dei requisiti sono stati fissati

Coherent I requisiti sono stati classificati ed i requisiti essenziali sono ben definiti

Acceptable I requisiti fissati definiscono un sistema soddisfacente per gli stakeholder

Addressed Il prodotto soddisfa i principali requisiti, al punto di meritare rilascio ed uso

Fulfilled Il prodotto soddisfa abbastanza requisiti da meritare la piena approvazione degli stakeholder

Progettazione

È una fase che precede la produzione e persegue la **correttezza per costruzione** piuttosto che la correttezza per correzione (conosciuta anche come "Trial And Error")
Progettando imparo a:

- Dominare la complessità del prodotto (Tramite divide-et-impera)
- Organizzare e ripartire le responsabilità di realizzazione
- Produrre in economia (Efficienza)
- Garantire la Qualità (Efficacia)

Dall'analisi alla progettazione

Enunciazione del Problema $\Longrightarrow$ Requisiti del problema $\Longrightarrow$ Soluzione del problema.
L'analisi comprende le prime due fasi di questo schema, mentre la progettazione e codifica comprende quella centrale e l'ultima.
Durante l'analisi faccio uso di un *approddio investigativo*, ricavando molti requisiti e molte soluzioni possibili.
Tra le varie soluzioni possibili, tramite un *approccio sintetico* scelgo quella con massima economicità

Obiettivi Della Progettazione

Gli obiettivi della progettazione sono:

- Soddisfare i requisiti tramite un sistema di qualità
- Definendo l'**architettura** (Strumento che permette di raggiungere un risultato) logica del prodotto

Architettura (Una definizione)

- Dividere il sistema in componenti
- Organizzazione di tali componenti (definizione di ruole, responsabilità, interazioni)
- Interfacce necessarie all'interazione tra componenti e tra componenti ed ambiente
- Paradigmi di composizione

Inoltre esistono diversi stili di architettura, tra cui dovremo scegliere quello che si adatta meglio al nostro problema

Qualità d una buona architettura

Sufficienza Soddisfa tutti i requisiti

Comprensibilità Comprensibile da parte degli stakeholder

Robustezza Capace di sopportare ingressi diversi (anche imprevisti o sbagliati)

Modularità Suddivisa in parti chiare e distinte:
È necessario dividere il tutto in parti che siano **utili**.
Inoltre perseguo l'information hiding, in modo da ridurre le perturbazioni esterne causate da cambiamenti interni.

Ancora sulle qualità di una buona architettura

Flessibilità Permette, al variare dei requisiti, di fare modifiche con costi contenuti

Riusabilità Le sue parti possono essere usate (utilmente) in altre applicazioni (in futuro)

Efficienza nel tempo, nello spazio, nelle comunicazioni (fra le parti)

Affidabilità Se un modulo fallisce non perdo il lavoro fatto

Disponibilità Vi è poco (o nessun) tempo di indisponibilità a causa di manutenzione (non tutto il sistema deve essere interrotto se qualche parte finisce sotto manutenzione)

Safety Resiste ai malfunzionamenti/Esente da malfunzionamenti gravi

Security Sicurezza rispetto alle intrusioni

Semplicità All'interno dell'architettura vi è solo il necessario e nulla di superfluo (meglio avere soluzioni semplici che non complesse)

Incapsulazione Non esporre l'esterno ad informazioni non utili, così da poter cambiare l'interno senza dover cambiare l'interfaccia (Information Hiding)

Coesione Dentro il modulo stanno cose che perseguono lo stesso obiettivo. Se manca una parte, il modulo risulta incompleto

Basso Accoppiamento Le parti dipendono poco o nulla tra di loro

Information Hiding

Le componenti sono come delle "scatole nere" ed i clienti ne conoscono solo l'interfaccia.
I funzionamenti interni delle componenti sono nascosti.
L'Information Hiding comporta alcuni benefici:

- L'esterno non può fare assunzioni sull'interno delle componenti
- Vi è una migliore manutenibilità
- Si hanno meno dipendenze ed aumentano le possibilità di riuso

Coesione

Le Funzionalità "vicine" devono stare nella stessa componente.
Ogni componente che sta in un modulo ha un motivo valido per starci.
Ha forti legami con il principio SOLID.

Tipi di Coesione Buona

- Funzionale
- Sequenziale
- Informativa

Ma su tutte prevale quella che persegue l'Information Hiding

Accoppiamento

Dipendenza reciproca "cattiva".
Si viene a creare:

- Facendo assunzioni sull'interno di certe cose (dall'esterno di tali)
- Imponendo, dall'esterno, vincoli sull'interno di una componente
- Condividendo frammenti di risorse

L'accoppiamento deve essere quanto più possibile minimizzato, ma non può essere portato a zero.

Misurare L'accoppiamento

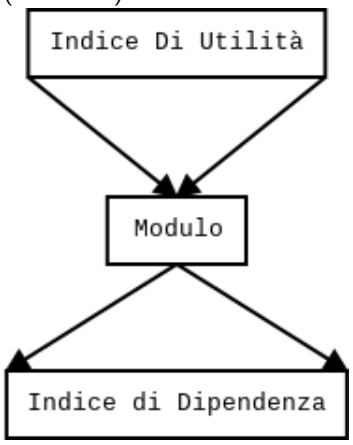
Considerando:

$$U = \text{utilizzo reciproco di M componenti}$$

Possiamo vedere che

$$U = \begin{cases} 0 \text{ come valore minimo} \\ M \times M \text{ come valore massimo} \end{cases}$$

Possiamo dire che più un modulo viene chiamato, più questo può essere considerato utile. Chiamiamo questa misura "indice di utilità" o "fan-in" ($SFIN$)
Mentre più un modulo fa uso di altri pezzi, più alta è la sua dipendenza da altri, chiamiamo questa misura "indice di dipendenza" o "fan-out" ($SFOUT$)



Riuso

Tramite riuso si capitalizzano sottosistemi già esistenti:

- Impiegandolì per più prodotti
- Ottenendo un minor costo realizzativo (sono già fatti)
- Ottenendo un minor costo di verifica (sono già testati e verificati)

È possibile

- Progettare **per** riuso (Più complesso, dato che bisogna riuscire ad anticipare i bisogni futuri)
- Progettare **con** riuso (Minimizzando le modifiche alle componenti riusate, altrimenti si perde il valore del riuso stesso.)

Costo puro nel breve periodo, risparmio nel medio termine (quindi può essere classificato come un investimento)

Progettazione Architetturale

Tre stili:

- Top-Down (Stile funzionale)
- Bottom-Up (Stile Object-Oriented)
- Meet-in-The-Middle: Approccio intermedio / misto.
  È quello usato più di frequente

Framework

Insieme integrato di componenti software prefabbricate:
Prima della programmazione object-oriented erano detti librerie.
Possono essere classificate (secondo la progettazione architetturale):

- Come Bottom-Up, dato che il codice è già sviluppato
- Come Top-Down perchè impongono uno stile architetturale più o meno specifico.

I framework sono molto utili come base riusabile di diverse applicazioni in un dato dominio.
Un esempio di framework può essere Swing, usato per le GUI in Java.

Pattern

Un pattern è uno stile o una serie di elementi ricorrenti in un'architettura.

Design Pattern Architetturali

Soluzioni progettuali a problemi ricorrenti.
Un paio di esempi di design pattern architetturali possono essere:

Architettura Multilivello In cui ogni strato conosce e comunica solo con gli strati adiacenti (Ad esempio in ISO/OSI o TCP/IP)
Un esempio di architettura multilivello è l'architettura "three-tier", composta da:

- Interfaccia Utente
- Logica di Applicazione
- Modello dei dati e persistenza

Architettura Model-View-Controller

Progettazione di dettaglio: Attività

Definizione delle **unità** realizzative (moduli), con l'obiettivo di ottenere specifiche da dare ai programmatori.
Successivamente si attua una specifica delle unità come moduli (a seconda delle caratteristiche del linguaggio utilizzato)

Progettazione di Dettaglio: Obiettivi

- Definire le unità architetturali allo scopo di realizzare le componenti dell'architettura logica
- Produrre la documentazione necessaria alla specifica di ogni unità (la cui struttura può seguire quella consigliata nello standard IEEE 1016:1998)
- Definire gli strumenti per le prove di unità (come casi di prova e componenti ausiliarie per verifiche di integrazione)

Stati di progresso secondo SEMAT

Architecture Selected Ho pensato all'architettura, al riuso, alle decisioni su make/buy/build

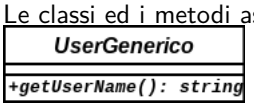
Demonstrable So enunciare e difendere (dimostrare) le proprietà buone della mia architettura

Usable Il Sistema è utilizzabile e la quantità di difetti residui è accettabile.

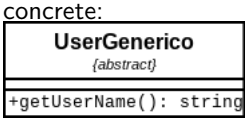
Ready La documentazione per l'utente è pronta e gli stakeholder vogliono che il prodotto divenga operativo

Classe Astratta

Non istanziabile.
Ha alcuni metodi astratti (Senza Implementazione), mentre altre operazioni possono avere implementazione.
Le classi ed i metodi astratti vanno scritti in corsivo.



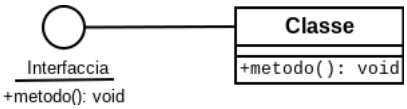
Dato che a mano è difficile distinguere tra corsivo e normale, è possibile fare uso delle proprietà aggiuntive per distinguere le classi astratte da quelle concrete:



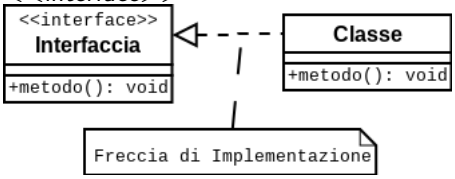
(Nota: Le proprietà aggiuntive vanno messe **tra parentesi graffe** e non doppie parentesi angolari)
Quando si va ad implementare un metodo in una classe concreta, questo **va scritto** (al contrario dei casi di ereditarietà)

Interfaccia

Classe priva di implementazione.
Una classe realizza un'interfaccia se ne implementa le operazioni. Negli schemi UML 2.x le interfacce sono rappresentate con un "lecca lecca":



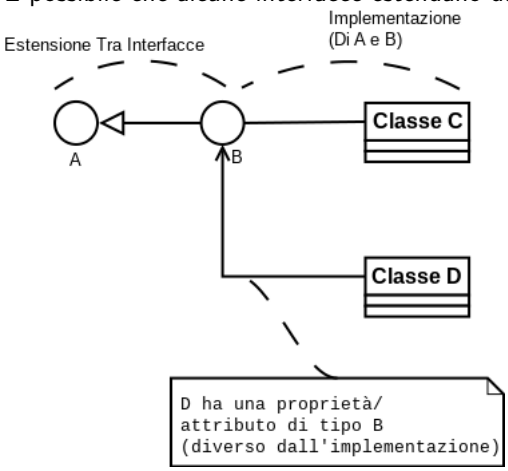
Mentre in UML 1.x si fa uso della parola chiave nativa di UML <<interface>>



Essendo una parola chiave nativa (e non una proprietà aggiuntiva) interface va scritto **tra parentesi angolari doppie**.
I metodi delle interfacce sono sempre pubblici.

Estensione Tra Interfacce

È possibile che alcune interfacce estendano da altre interfacce:



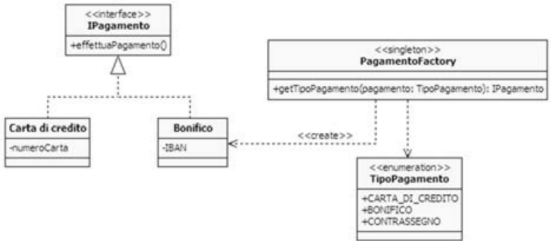
Attenzione

Estensione? Implementazione?

Per sapere se qualcosa implementa o estende un'interfaccia, è sufficiente fare riferimento ai significati di Java di "Extends" ed "Implements". Un'interfaccia può estendere un'altra interfaccia (ereditarietà) oppure una classe può implementare un'interfaccia (realizzazione)

Creazione di Istanze

Se ho uno schema in cui si va a creare un'istanza di una classe che implementa un'interfaccia, posso usare una forma contratta per rendere meno caotico lo schema. Quindi questa forma:

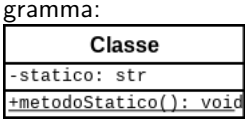


E questa:

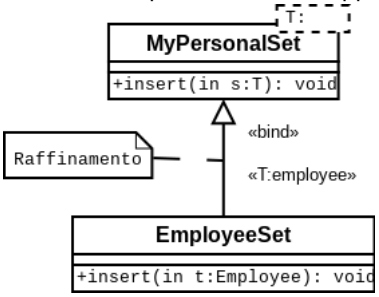
Vogliono dire esattamente la stessa cosa.

Varie

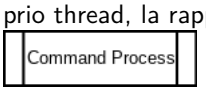
In UML, le operazioni e gli attributi statici vanno sottolineati nel diagramma:



Si possono aggiungere altri box all'interno delle classi, oltre agli attributi ed alle operazioni si possono avere box dedicati alle Eccezioni Lanciate (Exceptions) ed ai contratti delle classi (Responsibilities)
Inoltre si possono aggiungere parole chiave e proprietà aggiuntive, un esempio è:
<<enum>> che definisce classi enumeratore (tenere conto che ogni attributo di classi enum è pubblico.)
Inoltre vi è una parte dedicata ai generics/template/classi con tipo parametrico, che possono essere rappresentate in questo modo:



Inoltre è utile ricordare le classi attive, che eseguono e controllano un proprio thread, la rappresentazione in UML 2.x è:



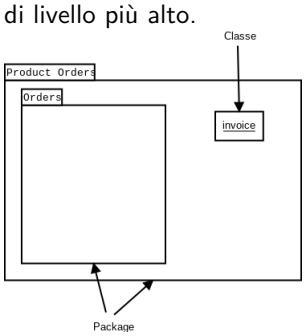
In UML 1.x si ingrossava il bordo del box "classe", ma questo rendeva solo lo schema più confuso.

Consigli

Si consiglia di evitare diagrammi troppo ricchi, in quanto sono poco utili, confondono le idee e troppi dettagli portano a troppi concetti da modificare nel caso si andassero ad aggiungere feature o modifiche di codice.
È opportuno iniziare esponendo concetti semplici.

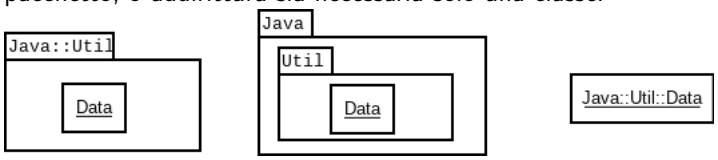
Diagrammi dei Package

Sono Un modo di raggruppare elementi UML (solitamente classi) in unità di livello più alto.

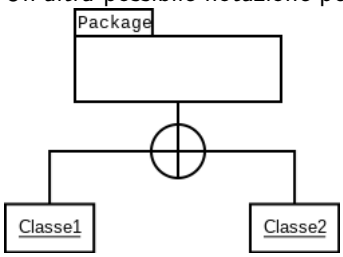


Dei package possono essere contenuti in altri package, con l'unica condizione che **non vi siano condivisioni tra package** (una classe può stare in un solo package), ogni package determina un namespace, quindi si possono avere situazioni con ugual nome, ma in package diversi; tipo:

Inoltre È possibile far uso dei nomi completamente qualificati per semplificare lo schema, soprattutto nel caso non si debba far uso di tutto il pacchetto, o addirittura sia necessaria solo una classe.

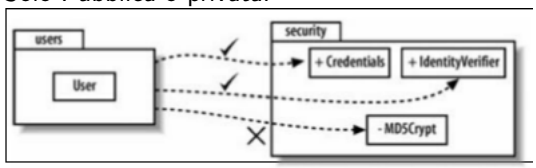


Un'altra possibile notazione per I package in UML 2.x è:



Visibilità nei package

Solo Pubblica o privata:



Interfaccia di un package

Insieme dei metodi pubblici delle classi con visibilità pubblica che stanno all'interno di un certo package.

Principi di progettazione

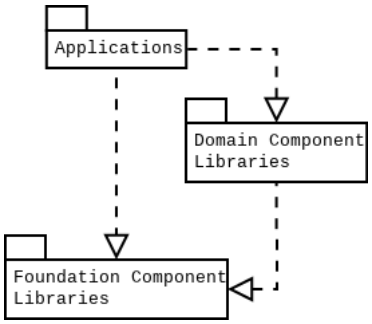
Quando si tratta di progettare i pacchetti, solitamente si fa riferimento ad uno di due principi:

Common Reuse Principle Si mettono nello stesso package le classi che solitamente sono riusate insieme

Common Closure Principle Si mettono nello stesso package le classi che hanno lo stesso obiettivo

L'uso di un principio o l'altro è questione di preferenze, ma solitamente si preferisce quello che permette di disegnare package più piccoli.
Il principio **common reuse** (solitamente) rende più semplice individuare ed attuare modifiche tra classi che presentano delle interdipendenze

Il diagramma



Il diagramma documenta le dipendenze fra le classi ed aiuta ad identificare, evitare e "rompere" eventuali **dipendenze circolari**.
I diagrammi di package dovrebbero presentarsi come grafi aciclici.
Inoltre è importante ricordare che le relazioni di dipendenza **non sono transitive**: se ho una classe C che dipende da B che a sua volta dipende da A, non è detto che se modifico A debba modificare anche C.

Rompere le dipendenze circolari

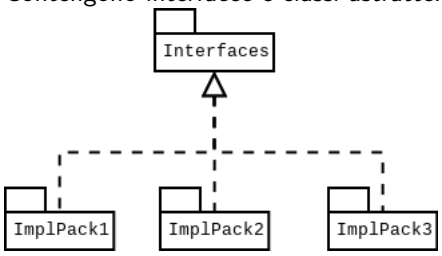
È possibile rompere le dipendenze circolare in 2 modi:

Fattorizzazione Creare un terzo pacchetto, da cui dipendono i primi due, che permetta di esternalizzare ciò che rendeva i due pacchetti interdipendenti

Riduzione Due pacchetti interdipendenti potrebbero essere prodotto di una fattorizzazione troppo forzata, ed i contenuti potrebbero in realtà stare meglio se sono nello stesso pacchetto

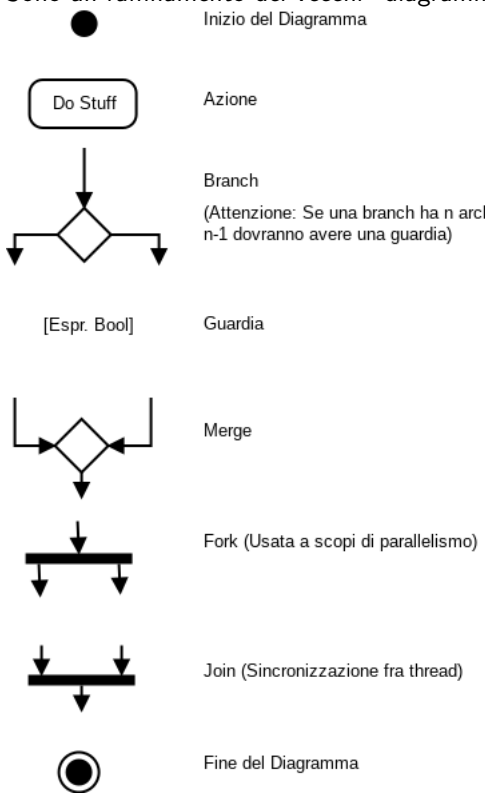
Package di Interfaccia

Contengono interfacce e classi astratte:



Diagrammi di Attività

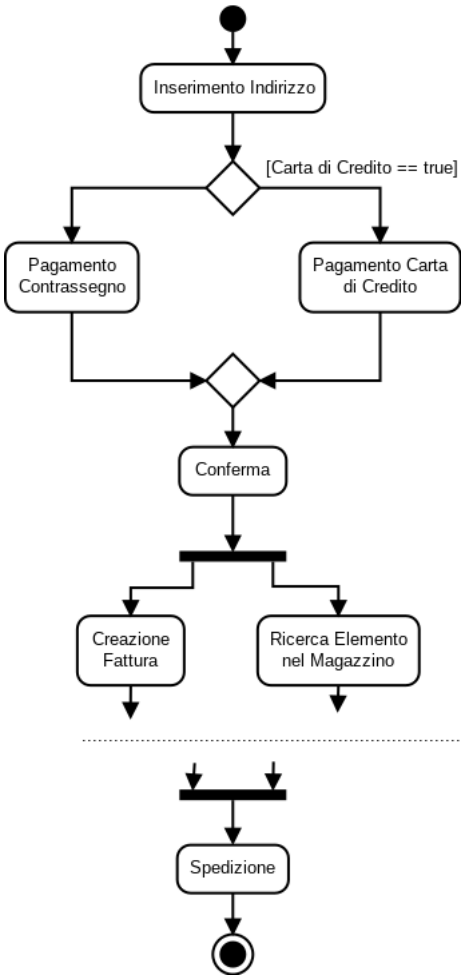
Sono un raffinamento dei vecchi "diagrammi di flusso:"



Consigli

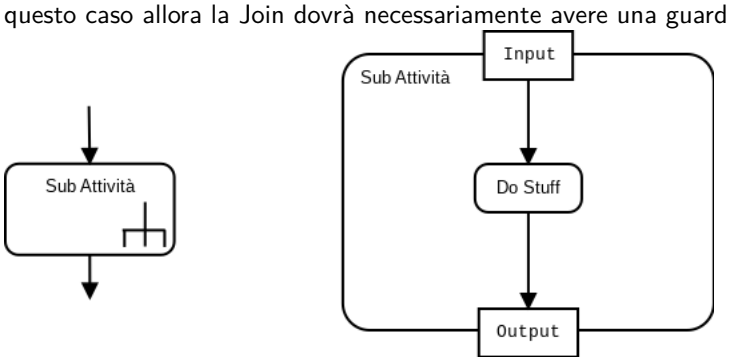
Evitare diagrammi troppo grandi, i diagrammi di attività vanno a modellare **un singolo processo**

Esempio: Ordine su Amazon



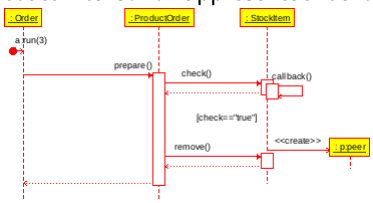
Nodo Di Fine Flusso

Usato per interrompere un thread senza necessità di usare una join, in questo caso allora la Join dovrà necessariamente avere una guardia



SubAttività

Permettono di dettagliare attività complesse, l'input ed output delle sottoattività sono rappresentati da dei rettangoli.



Parte mancata

Causa Ritardo

Fine dei diagrammi di attività

Diagrammi di sequenza

Descrivono la collaborazione di un gruppo di oggetti che devono implementare collettivamente un comportamento

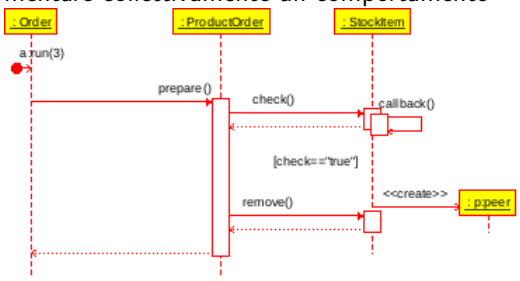
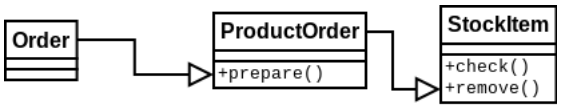


Diagramma delle classi corrispondente (Di Order, ProductOrder e StockItem)



Messaggi

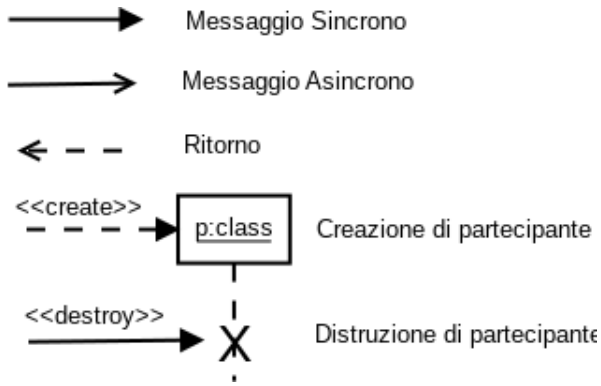
Dati e operazioni scambiati tra i partecipanti

- Chiamate a metodi di oggetti
- Messaggio trovato

Messaggi e Dipendenze

Gli attori possono inviarsi messaggi **solo se** tra loro vi è qualche dipendenza

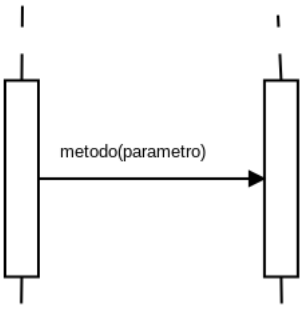
Tipi di Segnali/Messaggi



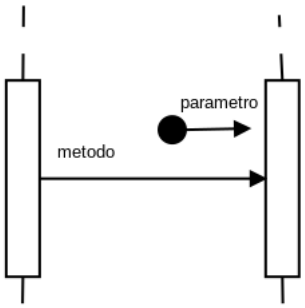
Passaggio di dati e parametri

Non vi è alcuna tecnica standard per il passaggio di parametri, solitamente si usano:

Metodo Classico:

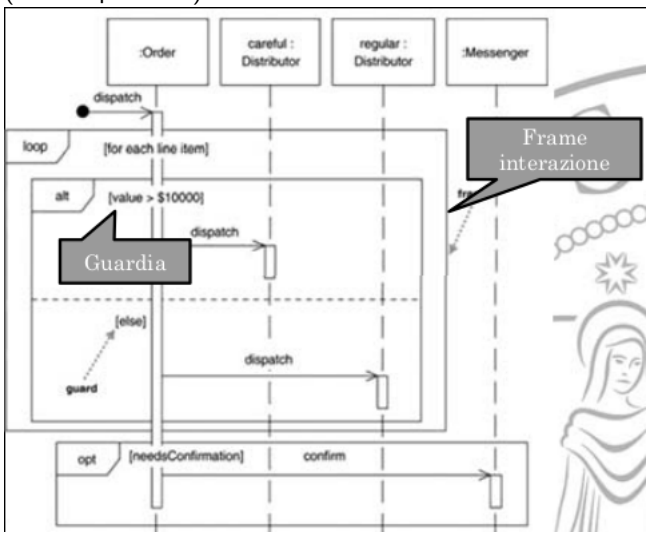


Notazione a girino (rende lo schema più pesante



Cicli e condizioni

Il diagramma di sequenza non sarebbe il più adatto per descrivere cicli e condizioni, ma si possono usare "frame di interazione", questi purtroppo tolgono il concetto di tempo che scorre dall'alto in basso del diagramma (cosa importante).



Diagrammi centralizzati vs Diagrammi distribuiti

Diagrammi Centralizzati Un attore controlla tutto il flusso di esecuzione, il che porta un oggetto ad avere troppe responsabilità; quindi in caso di cambiamento dei requisiti, questo oggetto dovrà probabilmente cambiare (Paradigma Procedurale)

Diagrammi Distribuiti Vi è un "rincorrersi di messaggi" tra più oggetti diversi (Paradigma Object Oriented)

Design Pattern Strutturali

Affrontano problemi riguardanti composizione di classi ed oggetti, consentono il riuso e sfruttano ereditarietà ed aggregazione.

Adapter

Converte l'interfaccia di una classe in un'altra.
Perchè? Perchè spesso i toolkit non sono riusabili (ad esempio librerie provenienti dall'esterno).
Questo pattern permette il riuso di una classe esistente che però non è conforme all'interfaccia target.
Permette la creazione di classi riusabili, anche con classi non ancora viste.
Da usare in caso non sia possibile riadattare l'interfaccia tramite l'ereditarietà (Vedi Object adapter)

Avviso

Lezione Mancata

Mancata flipped classroom sulla documentazione

Avviso

Lezione Mancata

Lezione mancata sulla qualità

ISO/IEC 14598

Fornisce il modello di valutazione, tramite **misurazione quantitativa**

Misurazione Quantitativa

Il processo tramite cui, secondo regole definite (e metriche), simboli o numeri sono assegnati ad attributi di una entità (N. Fenton)

ISO/IEC 25000:2005

Ingloba gli standard ISO 9126 e ISO 14598, chiamato anche
SQuaRE - Software Product Quality Requirements and Evaluation

Metriche Software

Qualunque tipo di misura che si relaziona ad un sistema Software:

Programmi : SLOC (Source Line of Code), cioè uno statement

Impegno : Giorni/Persona, Ore/Persona

Testo : Gunning's Fog Index
Fog = ((media di parole per frase)+(numero di parole con più di 3 sillabe)) · 0.4

Le metriche aiutano a misurare attributi, oltre ad aiutare a fare previsioni ed identificare anomalie.

Esempi di metriche

- Numero di Parametri di procedura (Meno è meglio)
- Complessità Ciclomatica (Meno è meglio)
- Dimensione del programma, in numero di linee di codice (Solitamente Meno è meglio, a seconda del linguaggio di programmazione)
- Numero di messaggi d'errore (Nei Log) (Solitamente di più è meglio, aiutando la manutenibilità ed aumentando il grado di diagnostica interna)
- Lunghezza del Manuale Utente (Solitamente più lungo e preciso è meglio)

Valutazione

Prodotto $\longrightarrow$ Misurazione (scelta delle metriche) $\longrightarrow$ Valutazione (Interpretazione delle misure) $\longrightarrow$ Accettazione (tramite confronto con i criteri di accettazione) $\longrightarrow$ Giudizio

Qualità Di Processo

Concentrarsi sulla qualità di prodotto non basta, e non dà risultati ripetibili, conviene concentrarsi sulla qualità del way of working.

- Organizzazione e diffusione interna **sistematica**
- Identificazione dei momenti di verifica in itinere
- Riproducibilità dei risultati
- Quality Assurance, anche **proattiva**

Bisogna inoltre avere **disposizione al miglioramento** (essere fieri del proprio way of working, ma sapere che è sempre migliorabile.)

ISO 900x

Certificazione ISO 9001 (Seconda metà anni '90)
Valutazione dei fornitori di prodotti o servizi.
ISO 9000:2005 - Fondamenti e glossario.
ISO 9001:2000 - Sistema Qualità - Requisiti (Praticamente ISO 9000 calato ai sistemi produttivi)
ISO 9000-3:1997 - Quality Management and Quality Assurance Standards
ISO 90003:2004 - Praticamente ISO 9001 applicato ai Software
ISO 9004:2000 - Guida al miglioramento dei risultati

Gestione Qualità Come Funzione Aziendale

Funziona trasversalmente a settori e repari e riferisce direttamente alla direzione

Manuale della Qualità

Documento che definisce il sistema di gestione della qualità di un'organizzazione

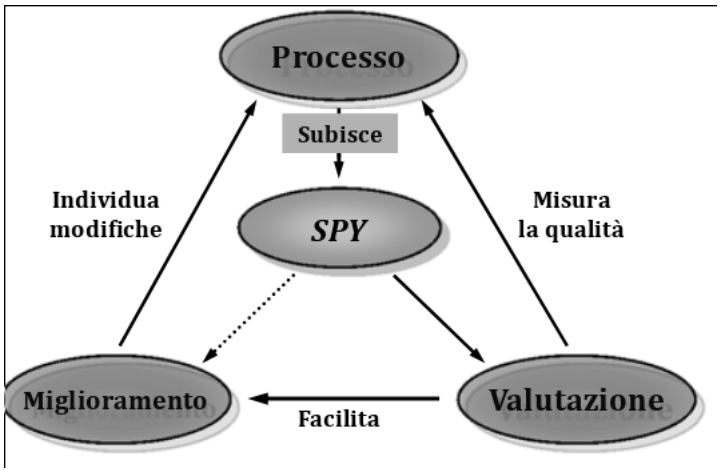
Strumenti di Valutazione

SPY Software Process Assessment And Improvement

CMM-CMMI Capability Maturity Model Integration

SPICE Software Process Improvement Capability Determination - Evolutosi poi nello standard ISO/IEC 15504

Modello SPY



CMMI

Capability Misura l'adeguatezza di un processo per gli scopi assegnati.
Caratteristica di un processo, preso da solo.
Determina l'intorno del risultato raggiungibile dal processo

Maturity Misura quanto (e quanto bene) l'azienda è governata dal suo sistema di processi.
È una caratteristica di un insieme di processi.
Risulta dall'effetto combinato delle capability dei processi considerati.

Model Criteri per valutare il grado di qualità dei processi dell'azienda

Integration Architettura di integrazione delle diverse discipline

Processi e Capability

Un processo a basso livello di capability:

- Dipende da chi lo attua
- È definito ed attuato in modo opportunistico
- Ha esito, avanzamento e qualità difficili da prevedere
- Porta a compromessi tra qualità e funzionalità

Mentre dei processi ad alto livello di capability sono eseguiti da tutti quanti in modo disciplinato, sistematico e quantificabile.

Governance

Intelligenza dei processi di una organizzazione (in termini di Efficacia, Efficienza, Manutenzione e Visione)
È un'attività continua

Livelli di Maturità CMMI

1. Initial - I processi sono imprevedibili, mal controllati e reattivi. Inoltre i processi sono non sistematici e non quantificabili.
2. Managed - I processi sono definiti per progetto, e sono solitamente reattivi (Qui applico "Do" del PDCA)
3. Defined - Processi sono caratterizzati per organizzazione e sono proattivi invece di reattivi (Si applica "Plan" del PDCA)
4. Quantitatively Managed - I processi sono quantificabili e controllati (Qui si applicano "Plan" e "Check" del PDCA)
5. Optimizing - Ci si concentra sul miglioramento, applicando completamente il PDCA

Produttività

Quantità di risorse usate per produrre una unità di prodotto **Conforme** (meno è meglio)

Costi e benefici del CMMI

- Si ha un aumento della produttività
- Miglioramento dell'identificazione dei difetti nelle prime fasi di produzione
- Riduzione del time-to-market
- Riduzione dei difetti rilevati sul campo
- Ritorno dell'investimento di circa 5 volte

ISO/IEC 15504

"Abolisce" la maturity basata sul "voto peggiore" (Il bottom) di CMMI e consente alle organizzazioni di vedersi con più sfaccettature, invece di focalizzarsi sul peggio.
È più complesso del CMMI ma consente di avere una visuale più granulare dello stato delle capability dei processi, uno ad uno.

Class Adapter

Non funziona quando bisogna adattare una classe e le sue sottoclassi (dato che fa uso di ereditarietà), ma permette all'adapter di modificare alcune caratteristiche dell'adaptee

Object Adapter

Permette all'adapter di adattare più tipi
Non permette la modifica di caratteristiche
Un oggetto adapter non è sottotipo di adaptee
Sono anche detti **Wrapper**

Implementazione

È importante individuare l'insieme minimo di funzioni da adattare

- Rende l'implementazione più semplice
- Rende la manutenzione più semplice
- Fare uso di operazioni astratte

Decorator

Aggiunge responsabilità ad un oggetto **dinamicamente**
Usato quando ci sono molte funzionalità che possono essere mischiate ed aggiunte ad un tipo base. Usato quindi:

- Il subclassing non può essere sempre usato (ad esempio vengono a crearsi un numero esponenziale di subclassi)
- Si vogliono aggiungere funzionalità **dopo** (o prima) dell'esecuzione della funzionalità base

Altre note:

- Consente più flessibilità rispetto alla derivazione statica
- Evito di creare "agglomerati di funzionalità", consentendo di creare classi più semplici
- Attenzione! Il decorator è diverso dalle componenti:
  I decorator non vanno usati nel caso in cui la funzionalità si basi su test di identità tra oggetti
- Negativo: Possibilità di proliferazione di classi piccole e simili:
  - Che rendono lo unit testing difficile
  - Sono difficili da comprendere fuori dal contesto del decorator
  - Ma sono facili da personalizzare

Facade

È un design pattern semplice, ma tende ad essere abusato.
Ha lo scopo di fornire un'interfaccia unica semplice per un sottosistema complesso.
Quindi si applica quando si ha bisogno di un'unica interfaccia semplice.
Consente:

- Disaccoppiamento sottosistema/client
- Stratificazione di un sistema

Conseguenze:

- Meno accoppiamento tra client e sistema
  - Permette di eliminare dipendenze circolari
  - Tempi di compilazione e building ridotti (evita la ricompilazione di tutte le componenti quando si va a modificare una componente dietro il facade)
  - Non nasconde completamente le componenti di un sottosistema
- Single Point of failure - Se modifico il facade, vado a "rompere" le dipendenze esterne, rendendole non funzionanti
- Sovradimensionamento delle facade

Altro:

- Implementabile come classe astratta
- Permette la gestione di classi provenienti da più sottosistemi
- Definizione di interfacce "pubbliche" e "private", dove un facade nasconde un'interfaccia "privata"
- Singleton Pattern: Si ha una sola istanza del facade

Proxy

Fornisce un surrogato di un oggetto di cui si vuole controllare l'accesso.
Solitamente usato per rinviare il costo di creazione di un oggetto al tempo d'uso effettivo.
Ha la stessa interfaccia dell'oggetto che va ad inglobare.
Le funzionalità dell'oggetto "inglobato" sono accedute attraverso il proxy

Tipi di Proxy

Remote Proxy rappresentazione locale di un oggetto in un diverso spazio di indirizzi (esempio: Stub della Java RMI)

Virtual Proxy creazione di oggetti complessi on-demand

Protection Proxy controllo degli accessi all'oggetto originale

Smart Pointer Gestione della memoria

Conseguenze:

- Introduce un livello di indirezione che può essere "farcito"
- Il remote Proxy nasconde dove l'oggetto reale risiede
- Il virtual proxy fa delle ottimizzazioni
- Il protection proxy definisce ruoli di accesso alle informazioni
- Possibilità di implementare sistemi copy-on-write

Introduzione alla flipped class

Dobbiamo capire quali milestone è necessario settare, e se settarle partendo dal loro significato, oppure settare semplicemente una data di calendario.
È inoltre necessario capire come arrivare a tali mileston, cioè definire l'organizzazione del lavoro.

Milestone

Punto di calendario a cui dò un significato, un valore di progresso. È un punto di controllo

Chi decide la milestone?

È un'assunzione di responsabilità libera, che cambia a seconda del gruppo o dell'entità che si prende in carico il progetto

Come pongo le milestone?

- A seconda del loro significato
- In un punto fisso del calendario

Devo settare le milestone tenendo conto prima del loro significato, ma senza trascurare la presenza di vincoli temporali imprescindibili

Le milestone nel nostro progetto

Considerando come riferimento le slides del 16 ottobre, schede SEMAT 1 e come data della Revisione dei Requisiti (deadline) il 16 gennaio 2018, abbiamo 4 fronti su cui lavorare:

- Way of working
- Requisiti
- Lavoro (Work nel SEMAT) (Project Management)
- Software System

Prendendo come inizio la formazione dei gruppi, nella linea del "way of working" dovremo avere una milestone "principles estabilished" appena dopo la formazione dei gruppi, in cui definiamo a quali obiettivi di norme puntiamo, i principi ispiratori, come cercare le nostre norme.
Successivamente vi sarà una fase di "Foundation Estabilished", dove si hanno le fondamenta del lavoro preparate.
Prima del 16 gennaio, il way of working dovrà essere "in use", con un tempo sufficiente a poter redarre, secondo tali regole, la documentazione necessaria, ad esempio.
Nella linea dei requisiti, i requisiti dovranno essere "acceptable" (accettabili dagli stakeholder) prima del 16 gennaio, con un intervallo di tempo sufficiente a consentire eventuali correzioni di rotta.
Il lavoro, (cioè il project management) dovrà essere allo stato "started" molto prima del 16 gennaio, tra la venuta di un "way of working in use" e la definizione dei requisiti come "acceptable".
Quindi la gestione del progetto deve essere allo stato "started" abbastanza tempo prima della deadline, in modo che si abbiano dei requisiti accettabili.
Il sistema software invece troverà la sua prima fase, architecture selected, **dopo** il 16 gennaio.

Verifica

Accerta che l'esecuzione delle attività attuate nel periodo considerato non abbia introdotto errori

- È un'attività continua
- Si concentra sui processi
- Eseguito ad ogni avanzamento intermedio meritevole di attenzione

Validazione

Accerta che il prodotto sia conforme alle attese

- Fatto **una volta solo** a fine progetto
- "confronta" il prodotto finale con i requisiti
- Si concentra sul prodotto

Forme Di Verifia

- Analisi Statica
  - Non richiede l'esecuzione del Software
  - Essenziale finchè il sistema non è completamente disponibile
  - Studia il codice e la documentazione
  - Verifica
    - Conformità alle regole
    - L'assenza di Difetti
    - La presenza di proprietà positive
- Analisi Dinamica
  - Richiede l'esecuzione del software
  - Avviene tramite "test"
  - Usata sia per verifica che validazione

Verifica dell'analisi dei requisiti

Fatta contro il capitolato ed il dominio del problema, richiede

Tracciabilità: I requisiti derivano dal nulla, dal capitolato, dal dominio o dal proponente?

Regole per l'analisi: Mi chiedo:

- Ho regole per l'analisi dei requisiti? Le ho applicate bene?
- Ho una numerazione che classifica i requisiti in modo non ambiguo?
- I requisiti sono associati alla fonte?
- I requisiti sono associati al diagramma dei casi d'uso?
- I requisiti sono chiari?
- Ho trovato tutti i requisiti?
- Ho capito bene i requisiti?

Più prove trovo a favore dei vari requisiti, meglio è.

Verifica della Progettazione

Per la progettazione logica Viene fatta contro i requisiti, tramite regole definite

Per la progettazione di dettaglio Viene fatta contro la progettazione logica

Testing

La "Validazione del Fornitore"
Analisi Dinamica
Ogni sezione di test deve subire analisi statica contro le sezioni di test precedenti

Unità

Si definisce "unità" la più piccola quantità di Software:

- Verificabile **da solo**
- Prodotta dal singolo programmatore
- La definizione è sempre intesa in senso architetturale
- Non si tratta di linee di codice, ma di entità di organizzazione logica

Può essere vista come un compito assegnato ad un individuo, compreso completamente da tale individuo, verificabile singolarmente ed indipendentemente. Tale compito è portato a termine in breve tempo (solitamente nel giro di ore).
Prima di una commit, il codice deve passare i test di unità, quindi abbiamo una situazione in cui il programmatore può testare il proprio codice, anche se i testi sono scritti da altri.

Analisi Dinamica

Se manca il main in una unità, vado a costruire un elemento detto "driver" (o "mock").
Se ho invece una componente all'interno dell'unità da testare che non voglio però che faccia parte del test, la sosituirò con uno "stub", che simula il comportamento di tale componente.
Driver e stub sono componenti "usa e getta".
Quindi formalmente:

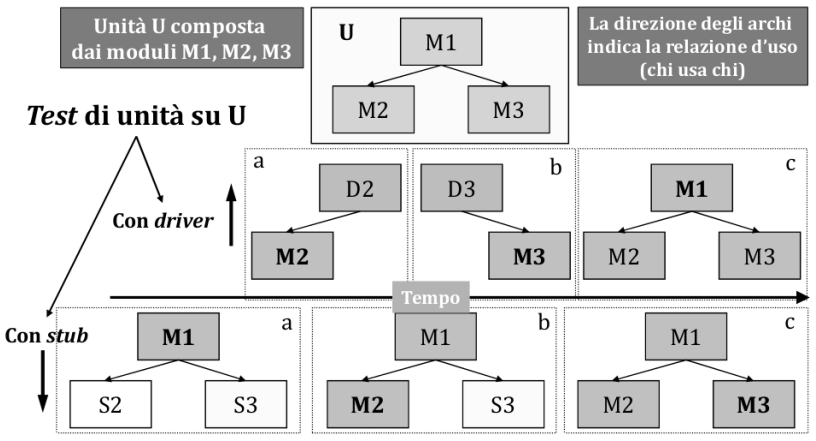
Driver Componente attiva fittizia che guida il test di una unità priva di main

Stub Componente passiva fittizia per simulare una parte del sistema non oggetto del test

Inoltre è vitale dotarsi di un'ulteriore componente:

Logger componente **non intrusiva** di registrazione dei dati d'esecuzione per analisi dei risultati

Esempio di Test



Tipi di Test

- Test di Unità
- Test di integrazione (tra le unità)
- Test di Sistema
- Test di regressione

Test di Unità

Come tutti i test, Sono attività di analisi **ripetibili e deterministiche**

- Col supporto di analisi statistica mirata
- Si svolgono con massimo grado di parallelismo
- Sono tanti, quindi vanno massimamente automatizzati
- Vanno effettuati sotto condizioni controllate

Test di regressione

Accertano che le modifiche fatte (che solitamente sono correttive) non facciano danno ad altri.
Quindi ogni parte P di S non deve causare errori in P od altre parti di S in relazione con P.
Vado a ripetere dei test già previsti ed effettuati per ogni P

Forme di Analisi Statica

L'analisi statica si fa "leggendo"
La lettura è fatta in 2 forme:

Walkthrough Attraversamento a pettine

Inspection Osservazioni fatte sulla base di sospetti

Walkthrough

- Cerca Difetti
- Esegue una lettura Critica
  - A largo spettro
  - Senza l'assunzione di presupposti
- Fatta da gruppi misti di ispettori e sviluppatori con ruoli ben distinti, ed un arbitro
- Per i listati di codice, vado a "simulare un'esecuzione"

Inspection

- Cerca difetti
- Esegue una lettura mirata, usando una checklist
- Costa meno del walkthrough

Attività di Quality Assurance

Raccolta di prove di qualità in maniera tempestiva

- A fronte di metriche ed obiettivi definiti
- Per controllo ed accertamento

Lo standard ISO/IEC 9126 è lo standard di riferimento per la qualità di prodotto.

Design Pattern Creazionali

Danno indicazioni per risolvere problemi di istanziazione

Singleton

(Diventato un anti-pattern)
Assicura **L'esistenza di un'unica istanza** della classe e dà un punto di accesso globale ad essa.
Applicabilità

- Deve esistere una sola istanza di una classe in tutta l'applicazione e deve essere accessibile in modo noto
- L'istanza deve essere estendibile tramite ereditarietà

Non voglio fare uso di variabili globali perchè:

- Introducono dipendenze
- Divengono un "single point of failure"

I singleton sono classi prive di stato, che contengono solo metodi e nessun attributo dipendente dall'istanza. Se i singleton fossero classi con stato, si incorrerebbe in fenomeni di Aliasing.
Si fa riferimento a "Singleto" (S maiuscola), per parlare del pattern, mentre si usa "singleton" (S minuscola) per parlare dell'insieme di caratteristiche del pattern.
Noi faremo riferimento alla seconda denominazione, soprattutto all'interno dei diagrammi UML

Singleton: Implementazione

- Rendo privato il costruttore, così che l'oggetto non possa essere istanziato da altri
- Mantengo un riferimento statico all'istanza all'interno della classe
- Creo un metodo statico nella classe **che crea l'istanza se il riferimento memorizzato è null**, e restituisce l'istanza statica memorizzata

Questa implementazione presenta dei problemi, oltre che dei vantaggi:

- In caso di concorrenza sul metodo statico, potrebbero venirsi a creare due singleton, di cui uno mai usato (dangling reference)
- Non è semplicemente serializzabile
- È un'implementazione Lazy, l'istanza del singleton viene creata solo quando necessario; risparmiando risorse.

Il problema della concorrenza può essere risolto inizializzando il riferimento statico immediatamente, andando però a perdere la laziness.
Si possono avere ulteriori migliormanti facendo uso delle classi enumeratore.
In questo modo sarà il linguaggio stesso (Java, nella lezione) che ci garantirà l'unicità dell'istanza

Errori Comuni

UML e Singleton

Molte volte negli anni passati si sono raffigurati i singleton come classi con un riferimento a se stesse (tramite la freccia).
Questa rappresentazione è **assolutamente sbagliata**

Builder

Separa la costruzione di un **oggetto complesso** dalla sua rappresentazione
Permette di costruire oggetti complessi senza aver a che fare con costruttori grandi e situazioni di telescoping (tanti costruttori che richiamano un costruttore grande con diverse combinazioni di parametri opzionali o meno)

Builder: Implementazione

- Rendo il costruttore privato o package private
- Creo una classe "builder" con gli stessi attributi della classe da costruire ed un setter per ogni attributo ed un metodo per costruire l'oggetto finale.
  Ogni setter dovrà ritornare il riferimento `this` per permettere la concatenazione di setter.

In questo modo (almeno in Java), si vengono a creare così delle cosiddette "interfacce fluide"

Vantaggi Builder

- Permette di controllare opzionalità ed obbligatorietà di parametri (tramite ad esempio objects.nonNull())
- Facilita le modifiche della rappresentazione interna di un prodotto, basta costruire un nuovo builder.
- Aumenta l'incapsulazione
- Miglior controllo del processo di costruzione
  - Tramite la costruzione step-by-step
  - Accentrando la logica di validazione

L'accentramento della logica di validazione dentro al builder (ed al di fuori della classe costruita) va bene dato che Builder e classe costruita sono fortemente legate

Abstract Factory

Fornisce un'interfaccia per costruire famiglie di prodotti non collegate tra loro ma le cui componenti di una famiglia vanno usate assieme; il tutto senza andare a creare classi concrete

- Demanda la costruzione degli oggetti ad una classe terza
- In questo modo evito errori del tipo "Creo un bottone per mac su una finestra per windows".
  La factory creerà gli oggetti corretti per me.
- Ho un unico punto di decisione nel codice, rendendolo configurabile.
- Devo nascondere i costruttori all'esterno, permettendo la costruzione degli oggetti solo tramite le factory
  Le factory possono essere delle buone classi singleton.

Vantaggi/Svantaggi Factory

- Promuove la consistenza fra prodotti
- Aumenta la semplicità nell'uso di una famiglia di prodotti
- Attua l'isolamento dei tipi concreti (in quanto i client vanno solo ad usare interfacce)
- Diventa però complesso aggiungere nuovi prodotti alle famiglie

Lezione Mancata

1 Lezione Mancata

Conclusione sulla qualità

Attenzione

2 Lezioni Mancate

Design Pattern Comportamentali (Iterator, Command, Strategy, Observer, Template Method) e Dependency Injection

Pattern Model-View

- Per supportare diversi utenti con diverse interfacce
- Evitando duplicazione di codice
- Senza influenzare le componenti che forniscono funzionalità base

MVC

Model-View-Controller

Model

È il modello della realtà
Contiene la "business logic" del programma.
Definisce quindi il modello dei dati e le operazioni che si possono effettuare su questi.
Viene progettato tramite tecniche Object-Oriented (Design Pattern)
Notifica la View dell'aggiornamento del modello dati (Observer Pattern)

View

È L'interfaccia utente.
Contiene la "presentation logic" del programma
Contiene quindi la logica di presentazione.
Cattura l'input e delega l'elaborazione al controller.
L'aggiornamento può essere fatto secondo:

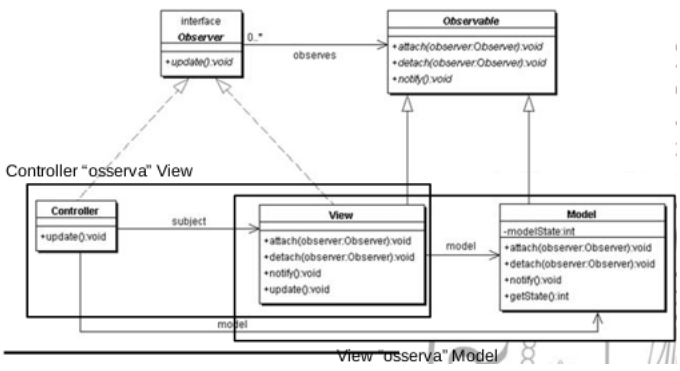
Push Model La view deve essere costantemente aggiornata (observer pattern);

Pull Model La view va a richiedere l'update quanto è opportuno.

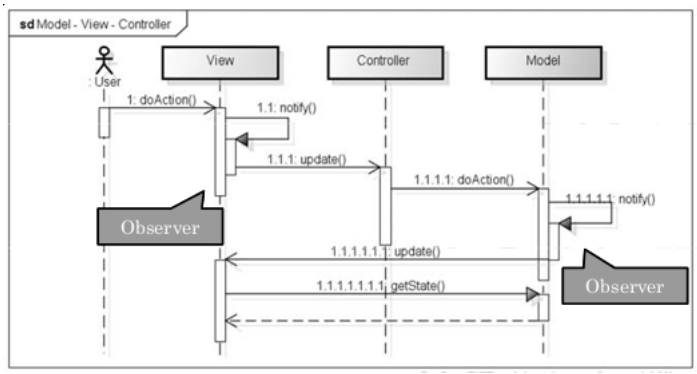
Controller

Contiene la logica di controllo tra model e view, cioè qual'è la reazione agli input dell'utente.
Contiene cioè la "application logic"
Trasforma quindi le interazioni utente in azioni sui dati.
Esiste solitamente un controller per ogni view.
Può implementare il strategy pattern (che modifica gli algoritmi che permettono l'interazione utente col model)

UML Classi del Push Model



UML Sequenza del Push Model



Conseguenze dell'MVC

- Evita che il team dedicato al front-end vada a modificare la business logic, così come l'inverso
- Permette parallelismo, separando i "concerns" (Disaccoppiamento)
- Permette il riuso delle componenti del model
- Supporto più semplice per nuovi tipi di client (Basta creare nuovi view e controller)
- Miglior Manutenzione e Testing
- Maggiore Complessità di progettazione (Ci saranno più classi, per garantire la separazione, ma tali classi avranno minori responsabilità)

MVP

Model-View-Presenter
Usato soprattutto su Android

View

Stiamo spostando responsabilità via dalla view, rendendola "stupida", privandola della presentation logic. Così diventa solo un template di visualizzazione (in Android è un file XML)

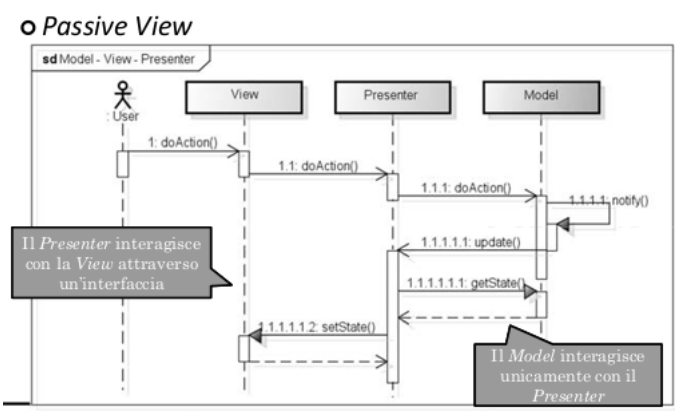
Presenter

Il presenter, tramite dei getter e dei setter, aggiorna e osserva la vista (fungendo da man-in-the-middle)

Conseguenze dell'MVP

Il presenter è facilmente testabile, facendo semplicemente un Mock della View

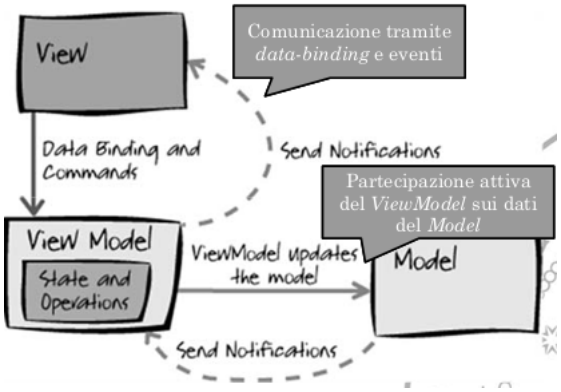
UML Dell'MVP



MVVM

Model-View-ViewModel
Separa lo sviluppo della UI da quello della business Logic.
La ViewModel è una proiezione del modello creata per una vista, nel modello resta solamente la validazione.
Viene eseguito un binding con la vista ed il modello tramite la ViewModel così che dati ed operazioni possano essere eseguiti su una UI.
La view Diventa dichiarativa (usando linguaggi di markup).
Avviene un 2-way data binding con proprietà del ViewModel e la view non possiede più lo stato dell'applicazione.
Tra View e ViewModel vi è un doppio observer pattern.
MVVM viene usato da AngularJS.

Schema Esplicativo

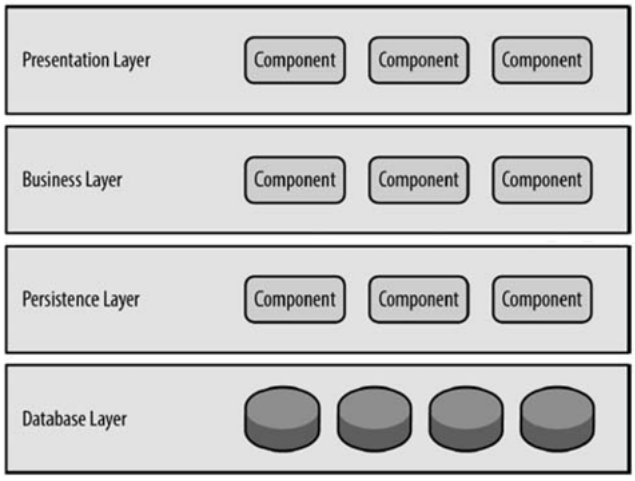


Problema

Accoppiamento (tra classi, componenti, . . .)
Nel caso l'applicazione non sia creata facendo uso di una struttura organizzata, questa risulterà molto difficile da scalare.
Inoltre se un'applicazione non è ben organizzata, questa non risponderà bene ai cambiamenti, rallentando di molto gli aggiornamenti alle ultime tecnologie (debito tecnologico). L'anti pattern dato dall'avere un insieme di moduli non organizzati viene detto *Big Ball of Mud*.

Layered Architecture

Solitamente usato per applicazioni molto piccole.
Le componenti sono organizzate "a strati", in modo orizzontale.

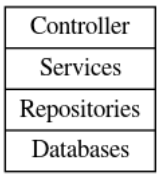


Caratteristiche della Layered Architecture

L'architettura di questo tipo più comune è la cosiddetta *3-tier architecture*, composta da Frontend, Business Logic e Persistenza.
Ogni strato ha una **singola** responsabilità.
Ogni strato parla solo con i layers adiacenti (imponendo quindi solo una dipendenza "verso il basso").
Le applicazioni sviluppate con questo stile sono semplici da testare, in quanto ogni layer può facilmente essere mockato.

Layers Chiusi vs Layers Aperti

Prendiamo per esempio un'applicazione Layered, che sia composta da questi quattro strati:



Nel caso si faccia una richiesta REST del tipo `GET professori/rcardin`, lo strato "services" fungerà solo da "passacarte".
In questo caso quindi si può adottare una struttura "a layer aperti", derogando ai principi layered classici e facendo in modo che il "controller" dialoghi direttamente con lo strato "repositories" (in questo caso si dice che il layer "services" è aperto).
Però:

- È un approccio pericoloso
- Introduce ulteriori dipendenze non controllate tra strati che non dovrebbero avere dipendenze
- È un'eccezione che dovrebbe essere usata solo se il guadagno dato da questo è **Molto Alto** (Si suggerisce di mantenere il "least astonishment principle").

Considerazioni

Buon Punto di Partenza È un pattern general-purpose solido

Architecture Sinkhole Anti-pattern Si rischia di avere a che fare con un anti-pattern in cui gran parte delle richieste passano attraverso strati che fanno solo da "passacarte". In tal caso si consiglia di aprire alcuni layer.

Scala male In quanto tende a creare un'applicazione monolitica.

Analisi del Pattern

Agilità	X	I piccoli cambiamenti sono isolabili, ma i grossi cambiamenti sono difficili a causa della natura monolitica del pattern.
Semplicità di Installazione/Deploy	X	Difficile per grandi applicazioni a causa dei deployment monolitici.
Testabilità	✓	Creare mock e stub dei layer è un'operazione semplice
Performance	X	Gran parte delle richieste attraversano molti layer, rallentando il sistema
Scalabilità	X	La Granularità troppo grossolana rende il sistema costoso da scalare.
Semplicità di Sviluppo	✓	È un pattern ben conosciuto, solitamente ha connessione diretta con la compagnia che produce il software.

Architettura Event-Driven

Un popolare pattern asincrono.

- È Molto adattabile
- Produce applicazioni molto scalabili
- I moduli di elaborazione degli eventi sono molto disaccoppiati ed hanno un obiettivo singolo
- Gli eventi sono processari in modo asincrono

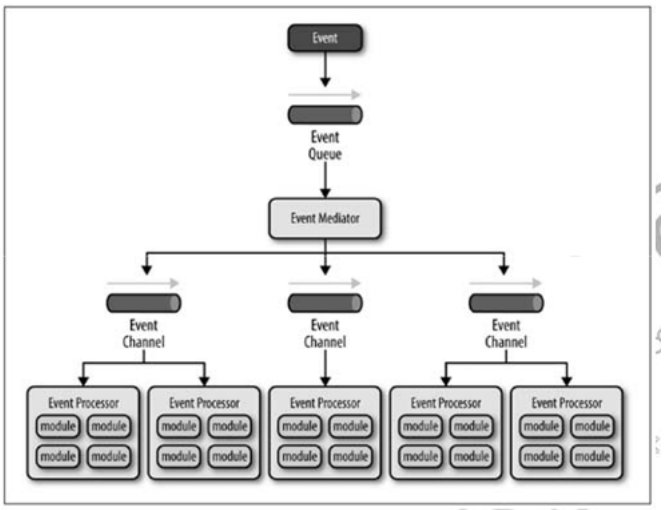
Esistono due topologie:

- Mediator Topology
- Broker Topology

Mediator Topology

Vi sono tante componenti con pochissime responsabilità ciascuna che hanno vita propria (e che non sanno nulla l'uno degli altri)
L'evento generato viene inserito in un componente che sa quali altre componenti singole chiamare, ed in che ordine. Tale componente è detta **Mediator** (o anche Orchestrator)

Schema Esemplificativo



Mediator

Il mediator **non effettua business logic** (trasformazioni, ad esempio.)
È fondamentale fare uso di un canale di comunicazione asincrono come:

- Una coda FIFO
- Un canale Topic (In cui si inserisce un messaggio e chiunque è in ascolto su tale topic può leggere)

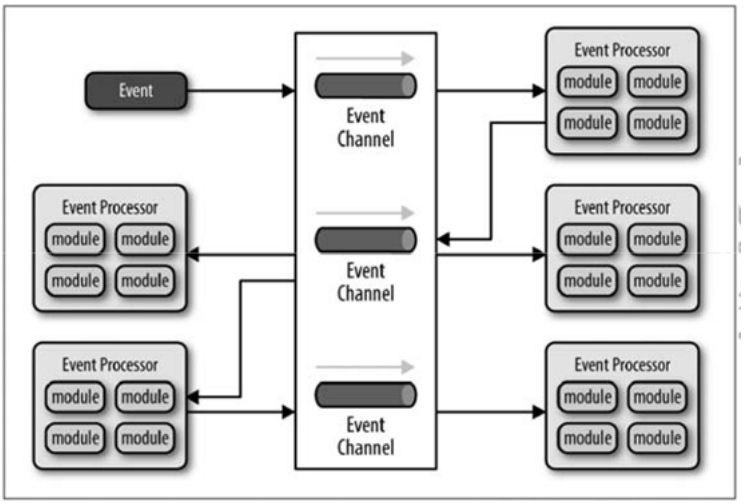
Event Processor

- Contiene la business Logic
- È responsabile della pubblicazione di nuovi eventi

Broker Topology

Non contiene alcun mediatore.
Ogni event processor legge un evento da un topic e pubblica un nuovo evento in un altro topic in modo diretto.
Il flusso per rispondere ad una richiesta è quindi distribuito.
Tramite questa topologia è semplice aggiungere moduli in ascolto sulle code, invece di andare a modificare il codice di un eventuale Mediator.

Schema Esemplificativo



Considerazioni

Le architetture ad eventi sono difficili da implementare.

È un'architettura completamente asincrona e distribuita Con i conseguenti problemi che affliggono i sistemi distribuiti: remote process availability, lack of responsiveness e reconnection logic. In ogni caso qualiasi cosa potrebbe andare storto.

Mancanza di transazioni atomiche Quali eventi possono essere avviati indipendentemente? Quale granularità mi è messa a disposizione per la loro gestione?

C'è bisogno di contratti forti per i processori di eventi Come ad esempio un formato standard per la comunicazione dei dati, come JSON o XML.

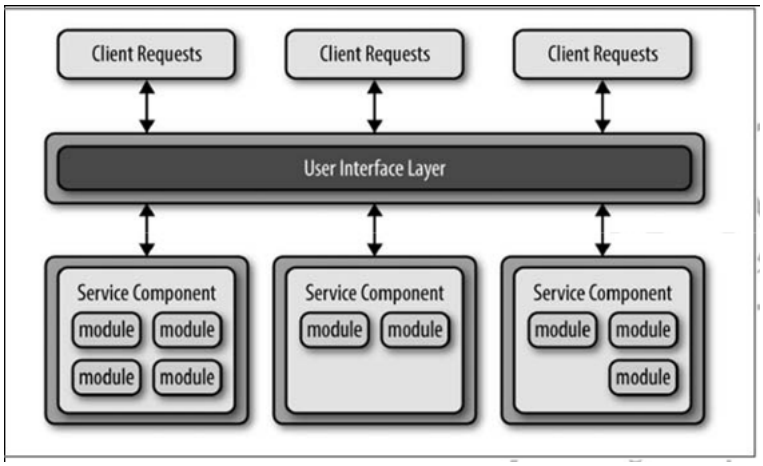
Analisi

Agilità	✓	I cambiamenti sono generalmente isolati e sono fatti in modo veloce e con poco impatto sul sistema
Semplicità di Installazione/Deploy	✓	Il deploy è semplice, data la natura disaccoppiata delle componenti. La broker topology è quella più semplice da questo punto di vista.
Testabilità	X	Richiede client specializzati per il testing che siano in grado di generare eventi.
Performance	✓	Alte performance, date le caratteristiche di asincronismo.
Scalabilità	✓	Gli event processor sono scalabili separatamente, dando un controllo granulare sulla scalabilità
Semplicità di Sviluppo	X	La programmazione asincrona richiede contratti forti e gestione avanzata degli errori

Architettura A Microservizi

Architettura ancora in fase di evoluzione.
Architettura fatta da tanti servizi, ogni servizio ha uno stack di persone dedicate alla gestione di tale servizio.
Il personale è quindi diviso in "Squad", in modo verticale (diversamente dal modo orizzontale dato dalle architetture layered)

Diagramma esemplificativo

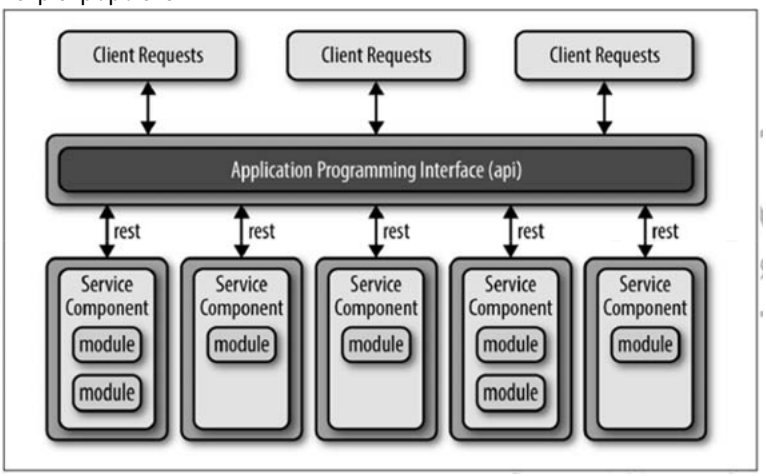


Problema

Nell'architettura a microservizi si viene meno al principio DRY (Don't repeat Yourself), quindi si avrà replicazione di codice allo scopo di mantenere la separazione completa tra i microcontrollori

Topologia API-REST

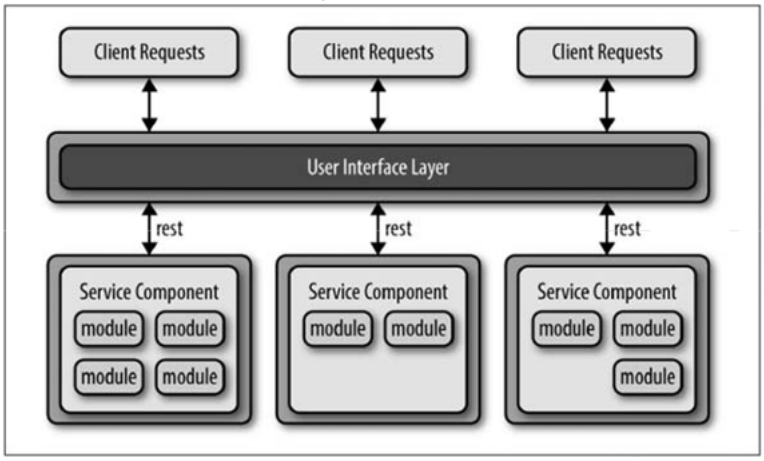
La più popolare.



REST si mappa molto bene col protocollo HTTP, quindi posso esporre un URL e fare uso di chiamate REST per effettuare operazioni tramite delle API.

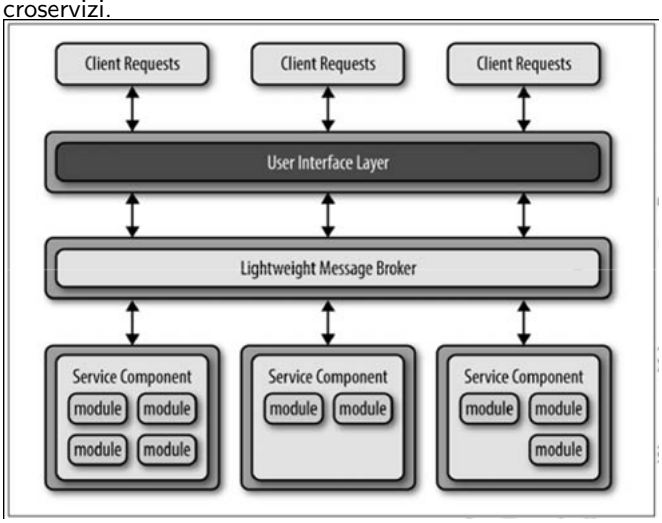
Topologia REST

Si fa uso di chiamate REST per accedere ai microservizi.



Centralized Message Topology

Invece di esporre più endpoint, ho una rappresentazione unica di più microservizi.



Analisi

Agilità	✓	Le modifiche sono solitamente isolate, il deploy è facile e veloce. L'accoppiamento è lasco.
Semplicità di Installazione/Deploy	✓	Il deploy è semplice a causa della natura disaccoppiata delle componenti. Hotdepoy e continuous delivery sono possibili.
Testabilità	✓	Grazie all'isolamento delle funzioni di business, il testing può essere reso specifico. Le possibilità di avere una regressione sono basse.
Performance	X	Data la natura distribuita del pattern, le performance non sono proprio alte.
Scalabilità	✓	I servizi sono scalabili separatamente.
Semplicità di Sviluppo	✓	Lo scope di business è piccolo ed isolato, richiedendo quindi meno coordinazione tra i developers ed i team di development.

Misurazione: Motivazioni e rischi

Perchè?

- Per conoscere e imparare
- Per valutare

Rischi connessi: Semplificazioni ed imprecisioni nelle misurazione possono provocare danni, anche gravi
La misura deve essere **oggettiva**, il che implica le seguenti caratteristiche:

- Ripetibilità
- Confrontabilità
- Confidenza (posso fidarmi della misurazione)

Misurazione: Definizione

Processo che assegna valori ad attributi di entità per descriverle secondo delle *regole definite*

Misura

- Il risultato della misurazione
- Assegnazione di un valore quantitativo ad un'entità, per caratterizzarne un attributo specifico.

Metrica

L'insieme di regole definite usate per la misurazione.

Indicatore

Misura sintetica usata per caratteristiche o fenomeni non misurabili direttamente.
Tramite gli indicatori (misurabili) posso ricondurmi ad una misura della realtà che non è direttamente misurabile.

Come si forma una valutazione

Serie Storiche (Tendenze nel tempo)
Posso popolare una serie storica quando i valori cambiano (altrimenti non avrebbe senso).
Per esempio per la serie storica riguardante delle lauree il popolamento potrebbe avvenire ogni anno

Benchmark Confronto con valori di riferimento derivanti dalle best practice (o best performance) del dominio
Quindi si va a puntare verso un obiettivo o riferimento (solitamente autorevole)

Adesione ad una checklist di regole od obiettivi definiti

Obiettivi di Misurazione

Valutare lo stato di

- Progetti (Stime, preventivi/consuntivi di tempi/costi)
- Prodotti (Misurando l'efficacia)
- Processi (Misurando la capability/maturity ed eventuali miglioramenti)
- Risorse (Misurando l'efficienza)

Tramite Attributi

- Interni: Che misurano l'azione/fenomeno in sé
- Esterni: che misurano il loro effetto o impatto

Per interpretare/cogliere tendenze In modo proattivo (invece di reattivo)

Per intraprendere azioni correttive Privilegiando quelle che danno maggior effetto col minor costo

Definizione Importante

Modello

Descrizione/Spiegazione della realtà, cioè spieghiamo il "perché" di una certa realtà

Processi di Misurazione

ISO/IEC 15939 - Software Measurement Process
Suddiviso in:

- **Measurement Information Model**
  Cosa misuriamo
- **Measurement Process Model**
  Come lo misuriamo

Information Needs

Sono basati su obiettivi, limiti imposti, rischi e problemi che si originano da processi tecnici e di gestione

Esempi di information needs

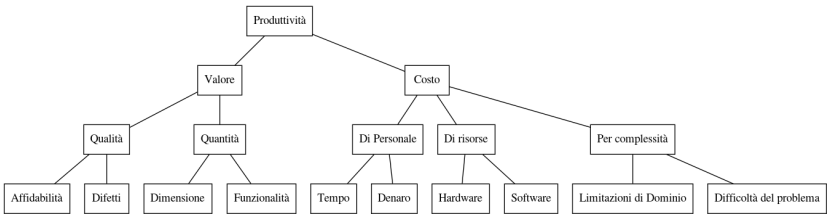
Per il management:

- Costo di un Processo
- Produttività degli sviluppatori
- Qualità del prodotto

Per gli sviluppatori:

- Verificabilità dei requisiti
- Copertura dei requisiti
- Quantità di difetti residui

Esempio: Modello di produttività



Lead Time

Tempo (di calendario) tra l'apertura (assegnazione "alla comunità") di un ticket e la sua chiusura con successo (in cui l'obiettivo è raggiunto).
È una metrica di gestione del progetto

Metriche Software

Problemi:

- Il software è immateriale ed è quindi difficile da misurare
- Il Software ha caratteristiche multiformi

Metriche di Progetto

- Grado di Coesione
- Grado di Accoppiamento
- Complessità Strutturale (Funzione del Fan-out)
- Complessità del flusso dati (Funzione del numero di parametri in ingresso ed uscita, oltre alla complessità strutturale)
- Complessità del sistema (Funzione della complessità strutturale e del flusso dati)

Metriche di Programmazione

- SLOC (Source Line of Code)
- Conteggio di costrutti/comandi

Complessità Ciclomatica

Complessità del flusso di controllo in funzione dei possibili cammini indipendenti all'interno di un singolo sottoprogramma.

- 1-10 Bassa
- 21-50 Alta
- >50 Inaccettabile

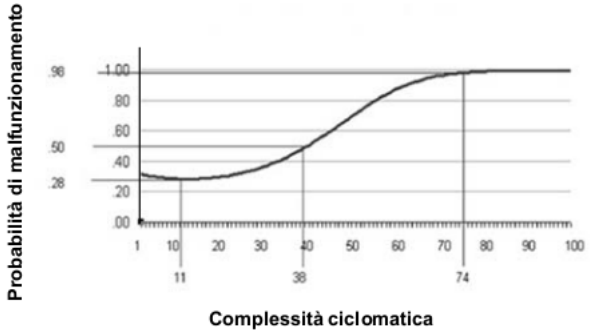
Fallibilità della complessità ciclomatica

Problemi rilevabili:

- Errore in difetto
  In caso di codice molto offuscato, si potrebbe avere una complessità ciclomatica bassa, ma il codice potrebbe risultare poco comprensibile.
- Errore in eccesso
  Ad esempio nel caso dello statement "switch", che viene conteggiato come una serie di if/elseif annidati, mentre invece risulta molto più leggibile di quest'ultimo.

Legame con la manutenzione

Nonostante la metrica sia fallibile, empiricamente è stato dimostrato che i programmi con complessità ciclomatica alta hanno problemi di manutenzione.



SOLID

Robert Martin (Il prof. Cardin consiglia il libro "Clean Code")
Dà dei dettami per avere un codice pulito, comprensibile e più possibile manutenibile.
Ovvero offre delle best practice per avere codice flessibile, robusto e riusabile.

S ingle Responsibility

O pen/Closed Principle

L iskov Substitution Principle

I nterface Segregation

D ependency Inversion

Solitamente i primi 3 sono considerati i più importanti

Single Responsibility

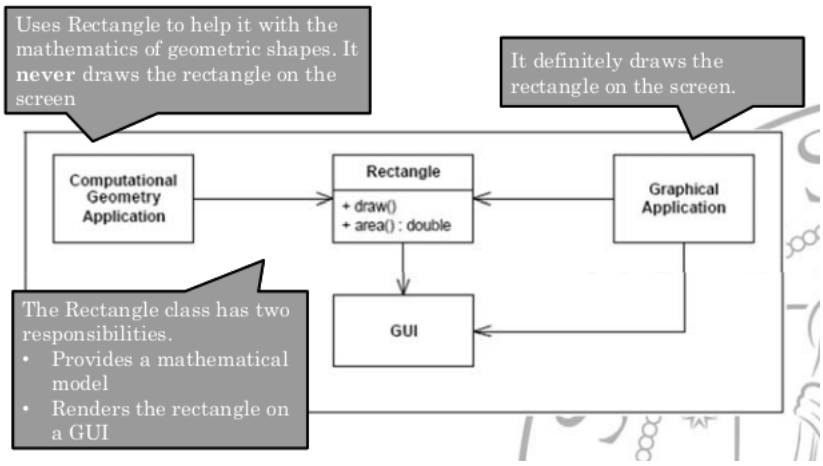
Basato sul concetto di **Coesione**.
Tipi che insieme vanno a comporre una funzionalità (o che comunque sono usati insieme), dovrebbero evolvere/cambiare insieme.
Un modulo dovrebbe avere una sola ragione per cambiare.
Concetto antitetico: **Accoppiamento** (dove si ha che due o più classi devono cambiare anche se non sono usate assieme)
In caso di cambiamenti ad una classe, l'accoppiamento ci forzerebbe a cambiare alcune (o tutte) le classi collegate

Memo

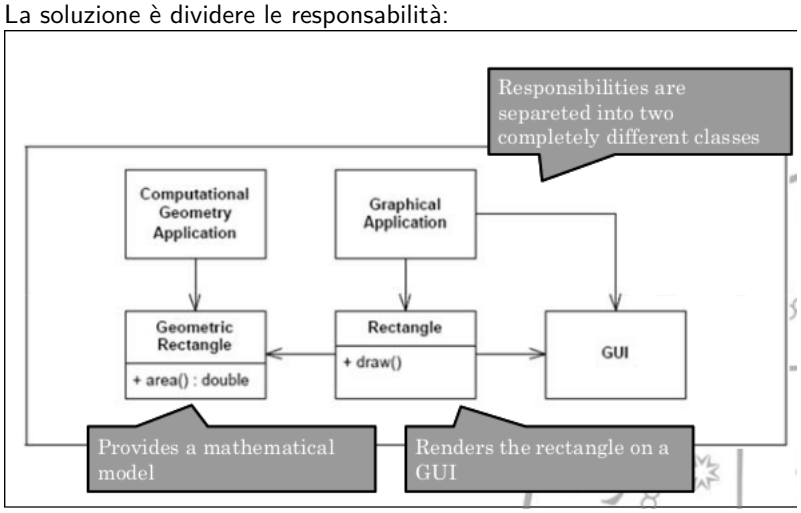
Dipendenze e Transitività

Le dipendenze **non sono transitive**
$A \longrightarrow B \land B \longrightarrow C \nRightarrow A \longrightarrow C$

Esempio

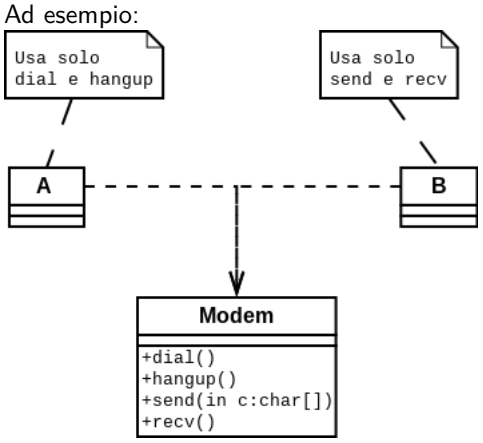


Alla "Computational Geometry Application" non dovrebbero interessare eventuali cambiamenti al metodo `draw()` di Rectangle, ma l'accoppiamento ci costringe ad effettuare comunque cambiamenti a "Computational Geometry Application" quando Rectangle viene aggiornato alla nuova versione.
La soluzione è dividere le responsabilità:



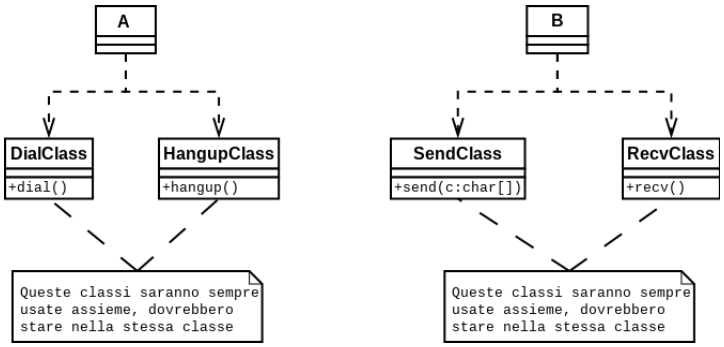
Cos'è una Responsibility

È un "asse di cambiamento" che va interpretato a seconda del contesto.
Ad esempio:



Tentativo

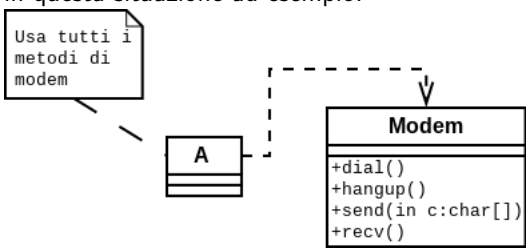
Proviamo ad applicare lo stesso principio usato in rectangle e separiamo i 4 metodi in 4 classi diverse. Ci ritroviamo con una situazione del tipo:



Ho perso coesione e mi ritrovo con degli accoppiamenti non previsti. In questo contesto la classe Modem aveva 2 assi di cambiamento (e non 4), cioè le funzione di "Gestione della connessione" (dial e hangup) e le funzioni di comunicazione (send e recv).

Assi di cambiamento

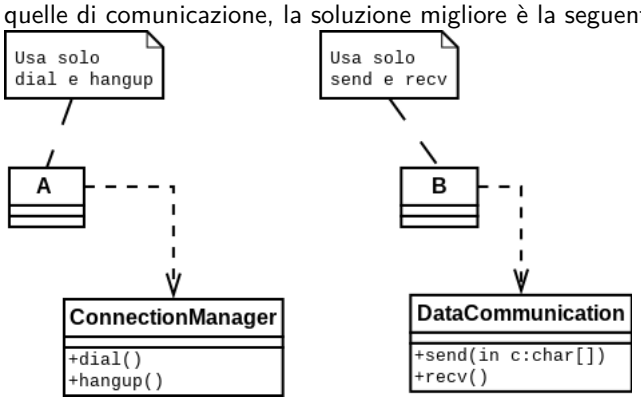
Ovviamente tale classe non ha due assi di cambiamento in senso assoluto, in questa situazione ad esempio:



La classe Modem è coesa e non è necessaria separazione delle responsabilità.

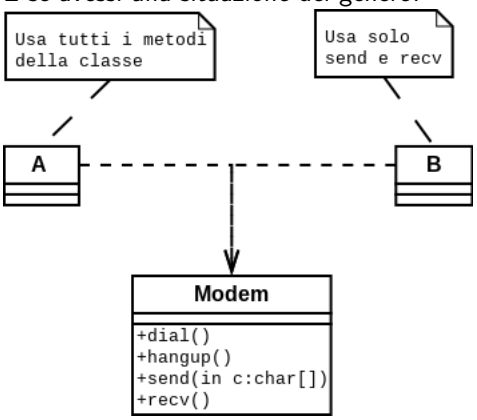
La soluzione corretta

Nel caso esaminato, in cui A usa le funzioni di gestione connessione e B quelle di comunicazione, la soluzione migliore è la seguente:

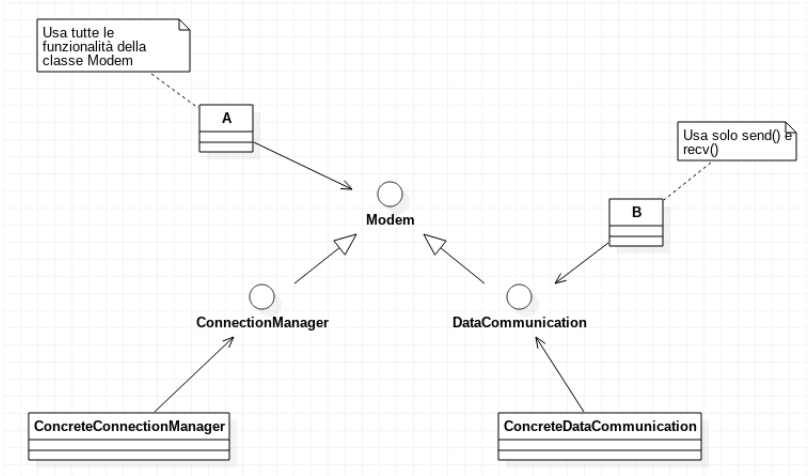


Domanda

E se avessi una situazione del genere?



Dobbiamo ricordarci che il principio funziona in casi astratti, quindi possiamo fare uso delle interfacce per avere il risultato migliore:



Open/Close Principle

Le entità software dovrebbero essere

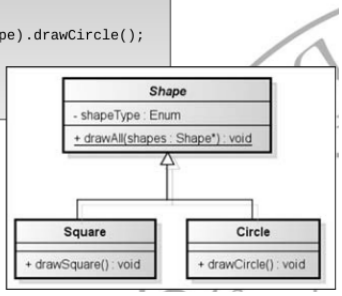
- Aperte all'estensione
- Chiuse alla modifica

I comportamenti di un software vengono quindi modificati **aggiungendo** codice e non modificando il codice esistente (a parte in caso di bug)
Questo principio può essere realizzato tramite **astrazione** (di tipo puro, cioè tramite interfacce).

Esempio

```
public static void drawAll(Shape[] shapes) {
    for (Shape shape : shapes) {
        switch (shape.shapeType) {
            case Square:
                ((Square) shape).drawSquare();
                break;
            case Circle:
                ((Circle) shape).drawCircle();
                break;
        }
    }
}
```

Does not conform to the open-closed principle because it cannot be closed against new kinds of shapes. If I wanted to extend this function, I would have to modify the function

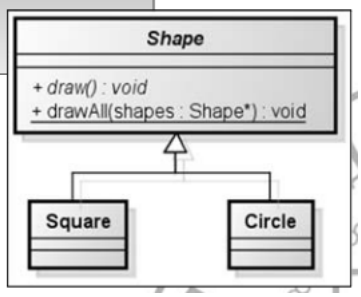


Se guardiamo il codice, DrawAll fa uso di cast, oltre alla presenza di un enum "shapeType" che limita l'estensibilità della gerarchia. Se voglio aggiungere una nuova forma, dovrò andare a modificare "Shape".

Riadattamento

È possibile riadattare la gerarchia all'open-close principle creando un metodo `draw()` astratto che viene sovrascritto dalle varie forme.

```
public static void drawAll(Shape[] shapes) {
    for (Shape shape : shapes) {
        shape.draw();
    }
}
```



Solution that conforms to open-close principle. To extend the behavior of the `drawAll` to draw a new kind of shape, all we need do is add a new derivative of the `Shape` class.

In questo modo evito le "cascate di cambiamenti"

Note sull'OCP

- Non è un principio assoluto: è impossibile avere un programma 100% chiuso, è necessario essere un po' strategici
- La chiusura si può ottenere per astrazione: tramite ereditarietà e polimorfismo
- La chiusura si può ottenere anche in modo "data-driven": usando strutture esterne.

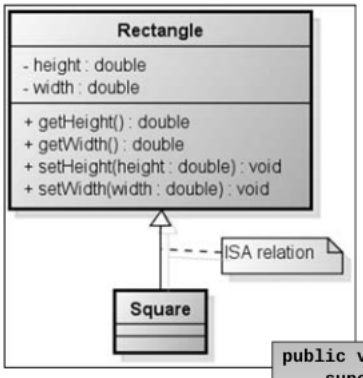
Dettami di OCP

Sono quelli standard della Programmazione OO:

- Le variabili membro devono essere tutte private
- Non si usano mai variabili globali
- La RTTI è pericolosa

Liskov Substitution Principle

Funzioni che usano puntatori o riferimenti a classi base, devono essere in grado di usare oggetti di classi derivate in modo trasparente.
Solitamente violare questo principio porta anche a violare l'Open/Close Principle.
Anche qui è necessario fare molta attenzione a come vengono usate le classi:



A `Square` does not need both `height` and `width` member variables. Yet it will inherit them anyway. Clearly this is wasteful.

`Square` will inherit the `setWidth` and `setHeight` functions. These functions are utterly inappropriate for a `Square`.
But, we could override them...

```
public void setWidth(double width) {
    super.setWidth(width);
    super.setHeight(width);
}
public void setHeight(double height) {
    this.setWidth(height);
}
```

Qual'è il problema

Dopo aver considerato questa gerarchia, vediamo il seguente test:

```
public void f(Rectangle r) {
    r.setWidth(42);
}
@Test
public void testF() {
    Rectangle r = new Square();
    r.setHeight(15);
    f(r);
    // This test will not pass!!!
    assertEquals(15, r.getHeight());
}
```

Il test non passa perchè nonostante Square sia un caso particolare di Rectangle, la precondizione di setWidth in square è più stringente di quella in Rectangle (infatti va ad effettuare anche un setHeight)
Se usate in questo modo, il codice del test richiede che Square e Rectangle siano due classi completamente separate.

Cosa se ne evince

La validità di un modello può essere valutata solo in funzione del modo in cui tale modello è usato.
Un altro consiglio è quello di non usare ereditarietà allo scopo di condividere del codice: ciò che veramente contano sono le operazioni che possono essere usate dai client esterni.
È possibile evitare questo problema facendo uso del "Design Per Contratti", che implica il dipendere da interfacce.
Quando vado a ridefinire una routine (in una derivata), posso sostituire la precondizione solo con una precondizione più debole e posso sostituire una Post-condizione soltanto con una Post-condizione più forte.
Questo sta alla base della "Programmazione per invarianti" (che è dispendiosa in termini di tempo)

Importante

Parte Saltata

Il professore ha ritenuto opportuno saltare la parte sulla Dependency Inversion e l'interface segregation

1 Ripasso sugli standard

Durante questo corso sono stati proposti molti standard che riporterò qui sotto per praticità di consultazione.

1.1 Standard Fondamentali

Gli standard qui sotto non devono assolutamente essere dimenticati per poter passare l'esame:

ISO/IEC 12207 *Processi di ciclo di vita* - Fornisce una visione ad alto livello dei processi coinvolti nel ciclo di vita di un Software (Primari, Organizzativi, di Supporto);

ISO 9000 *Modello di qualità di processo* - Neutrale rispetto al dominio applicativo;

ISO 9001 *Sistema di gestione qualità* - Praticamente ISO 9000 calato ai sistemi produttivi. È anche una certificazione;

ISO/IEC 9126 *Modello di qualità di un prodotto software*;

ISO/IEC 14598 *Linee guida per misurazioni e metriche di un prodotto software*;

ISO/IEC 25000 *SQuaRE: Software product Quality Requirements and Evaluation* - Ingloba in sè gli standard 9126 e 14598;

ISO/IEC 15504 *SPICE: **S**oftware **P**rocess **I**mprovement **C**apability d**E**terminaiton* - Misurazione e valutazione di capability dei processi;

ISO/IEC 15939 *Metriche di processo* - Fornisce un modello di misurazione ed un modello di processo di misurazione

1.2 Altri Standard

Questi standard sono collaterali ma collegati a quelli più importanti:

ISO/IEC 90003 *Linee guida per l'applicazione di ISO 9000 al software*

ISO/IEC 9004 *Guida al miglioramento dei risultati*